JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Traceability decision making in the case of the presence of conflicts

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Software Engineering

Eingereicht von:
Egor Erofeev

Angefertigt am:
Institut für System Engineering und Automation

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

Mitwirkung:
Dipl.-Ing. Achraf Ghabi

Linz, Oktober, 2013

Danksagung

Ich möchte mich an dieser Stelle bei allen Personen bedanken, die mich auf dem Weg zur Fertigstellung dieser Arbeit begleitet haben.

Hervorragende fachliche Unterstützung habe ich von Herrn Univ.-Prof. Dr. Alexander Egyed M. Sc., Herrn Univ.-Prof. Dr. Armin Biere, Herrn Dipl-Ing. Achraf Ghabi und Herrn DI(FH) Dr. techn. Alexander Nöhrer erhalten.

Des Weiteren möchte ich meiner Familie danken, die immer ein öffenes Ohr für mich hat und die mich während meines gesamten Studiums unterstützt hat.

Kurzfassung

Heutzutage sind Softwaresysteme mit großer Anzahl von kritischen Aktivitäten in verschiedenen Bereichen des täglichen Lebens verbunden. Traceability ist eine der Voraussetzungen für die Fähigkeit das hohe Qualitätsniveau dieser Software und kritischer Elemente jedes komplexen Systems zu erreichen. Traceability kann in vielen Bereichen wie Wirkungsanalyse von Änderungen, Change Management, Softwarevalidierung, Softwareverifikation, Testen und Softwarewiederverwendbarkeit eingesetzt werden.

Die TM (traceability matrix) ist eine der Methoden die Traceability zu erfassen und darzustellen. Die TM ist ein Dokument in tabellarischer Form, welches Artefakte des Softwareentwicklungsprozesses miteinander verbindet. Ein großer Nachteil ist, dass die Beziehungen in der Matrix in der Regel manuell erfasst werden. Für komplexe Systeme können diese fehlerhaft sein. Ein weiteres Problem ist die große Anzahl von Entscheidungen, die der Ingenieur / die Ingenieurin treffen muss.

Diese Masterarbeit liefert einen Ansatz und Algorithmen, die die Nachvollziehbarkeitsanalyse unterstützen und die Auswirkungen von Nachteilen der TM minimieren. Entwickler können mit sogenannten Dependencies (Abhängigkeiten) zwischen Gruppen von Artefakten arbeiten, anstatt direkte Beziehungen zwischen Artefakten zu identifizieren. Unsere Methode nimmt einen Satz von Dependencies am Eingang und setzt diese in eine Formel der Aussagenlogik in konjunktiver Normalform um. Diese Umsetzung ermöglicht die Schwerpunkte von SAT-Solvers für die Nachvollziehbarkeitsanalyse wiederzuverwenden. Benutzung von effektiven SAT-Solvers verbessert die Skalierbarkeit. Außerdem toleriert unser Ansatz Inkonsistenzen in der Benutzereingabe über Strategien um mit Inkonsistenzen während der Entscheidungsfindung umgehen zu können und wahrt die Analyseautomatisierung im Vorhandensein von Konflikten, sowie hilft dem Benutzer diese Konflikte zu lösen.

Die Evaluierung des Ansatzes zeigt, dass dieser eine gute Leistung bietet und daher für große Systeme eingesetzt werden kann. Zufolge der Dependenciessemantik liefert der Algorithmus korrekte Ergebnisse. Wie erwartet, steigert die Matrixabdeckung im Falle des Inkrementalfolgerns mit jedem Schritt bis zum Maximalwert. Die Isolationsstrategien weisen angemessene Effizienz, Recall und Precision in Bezug auf die Problemgröße

und Fehleranzahl nach. Bei fehlerfreien Problemen benötigt der Algorithmus nicht mehr als zwei Sekunden für die gesamte Traceanalyse.

Der Ansatz ist automatisch und hat die Toolunterstützung. Das Tool erlaubt die korrekte Traceanalyse mit dem SAT-Solver im Falle der forhandenen Konflikte und bietet die Benutzersuntertützung während des Konfliktlösungsprozesses. Die Isolation kann auf verschiedenen Granularitätsebenen durchgeführt werden, was die Arbeit mit komfortabler Detailgenauigkeit erleichtert.

Abstract

Nowadays software systems are close connected to a large number of critical tasks in various areas of daily life. Traceability is one of the prerequisites of the ability to reach the high level quality in this and similar software and thus the critical element of any rigorous system. The traceability can be successfully applied in areas like change impact analysis and change management, software validation verification and testing, software reuse, better artifact understanding, thereby increasing the quality and simplicity of software processes.

The TM (traceability matrix) is one of the methods of traceability recording and representation. The TM is a document in the form of a table that correlates artifacts within the development process in respect to the traceability relations. One of the TM's disadvantages is that the relations stored in the matrix are typically captured manually and may be error prone for large and complex systems. Another problem is a very large number of decisions that an engineer has to make.

This master thesis contributes an approach and algorithms to support trace analysis and minimize the impact of the trace matrix weaknesses. To make the traces establishing process easier, developers or engineers working with TM may provide dependencies between groups of artifacts instead of identifying trace relations between individual artifacts directly. The approach takes a set of such dependencies as input and transforms the input into CNF, what allows to reuse the effectiveness of SAT-solvers to performs the trace analysis and fill up cells of the trace matrix. Usage of effective SAT-solver supports excellent scalability. Additionally, the approach tolerates inconsistencies in the user input using an isolation technique and allows to perform analysis in case of the presence of conflicting dependencies as well as helps an engineer to resolve these conflicts.

The empirical evaluation of the approach shows that it provides good performance and, therefore, can be applied for large systems with more than 40K cells in the trace matrix. The algorithm produces correct results following the semantic of dependencies. As expected, in case of incremental reasoning, when a developer adds dependencies step by step, the coverage of the matrix increases with each step to the maximal value. The isolation strategies demonstrate reasonable efficiency, recall and precision depending on

the problem size and the number of errors. For error free problems, the algorithm requires maximal 2 seconds to complete the trace analysis even for very large problems.

The approach is automatic and tool supported. It allows correct trace analysis with SAT-Solvers in the presence of conflicts and uncertainties, continue working without any adaptions and provides the user support in conflicts resolving. The isolation may be performed on different granularity levels, what facilitates the work to the comfortable level of details.

Contents

List of Figures **ix**

List of Tables **x**

Abbreviations **xi**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Goals of this thesis . . . 3
1.3 Chapters description . . . 4

2 Traceability Basics **6**
2.1 Traceability relations . . . 6
2.2 Purpose of the traceability . . . 8
2.2.1 Traceability for change impact analysis and change management . 10
2.2.2 Traceability for software validation, verification and testing 11
2.2.3 Traceability for software reuse . . . 11
2.2.4 Traceability for artifact understanding . . . 11
2.3 Traceability retrieving, recording and maintenance . . . 12
2.3.1 Traceability retrieving . . . 12
2.3.1.1 Manual creation of traceability relations . . . 12
2.3.1.2 Semi-automatic generation of traceability relations 12
2.3.1.3 Automatic generation of traceability relations 13
2.3.2 Traceability recording . . . 14
2.3.2.1 Single centralized database approach . . . 14
2.3.2.2 Software repositories . . . 14
2.3.2.3 The hypermedia approach . . . 15
2.3.2.4 The mark-up approach . . . 15
2.3.3 Traceability maintenance . . . 15
2.3.3.1 The event-based approaches . . . 16
2.3.3.2 Rule-based approaches . . . 16
2.3.3.3 Approaches based on recognizing evolution . . . 16

3 Uncertainties And Conflicts In Traceability **17**
3.1 Uncertainties in the traceability . . . 17
3.1.1 A Scenario-Driven Approach to Trace Dependency Analysis 17
3.1.2 A Scenario-Driven Approach to Trace Dependency Analysis: Extention . . . 21

3.1.3 Traceability uncertainties between architectural models and code (Related work) . . . 25
3.2 SAT-based reasoning for the tracebility analysis . . . 34
3.3 Conflicts resolution strategies . . . 38

4 Approach **46**
4.1 Isolation strategies . . . 47
4.2 Oracle improvement . . . 51
4.3 Incremental reasoning . . . 56

5 Evaluation **62**
5.1 Test cases generation . . . 63
5.2 Test cases parameters . . . 67
5.3 Correctness and efficiency . . . 68
5.4 Isolation . . . 69
5.4.1 Recall . . . 71
5.4.2 Precision . . . 72
5.5 Scalability . . . 73

6 Tool **76**
6.1 Eclipse platform . . . 76
6.2 Tool architecture . . . 77
6.3 User interface . . . 78
6.3.1 Source code of problem description . . . 78
6.3.2 Model editor . . . 79
6.3.3 Visualization . . . 80
6.3.4 User guidance . . . 82
6.4 Third party libraries . . . 82

7 Conclusions And Future Work **84**
7.1 Summary . . . 84
7.2 Future works . . . 85
7.2.1 Multidimensional reasoning . . . 85
7.2.2 User guidance for units and cells . . . 85

A ANTLR Grammar **87**

Bibliography **90**

List of Figures

3.1 VOD Statechart diagram 18
3.2 Footprint graph 31
3.3 CNF 34

5.1 Coverage for GanttProject 69
5.2 Isolated dependencies vs. all dependencies 70
5.3 Isolated dependencies vs. all dependencies (cells level) 70
5.4 Different of coverages (cells level) 71
5.5 Recall 72
5.6 Precision 72
5.7 Precision (cells level) 73
5.8 Scalability (small problems) 73
5.9 Scalability (large problems) 74
5.10 Scalability (with errors) 74
5.11 Scalability (incremental reasoning) 75

6.1 Tool structure 77
6.2 Class diagram 78
6.3 Model Editor 80
6.4 Trace matrix 80
6.5 Footprint graph 81
6.6 Outline 81
6.7 Conflict resolving assistant 82

List of Tables

3.1 Test scenarios for VOD player 18
3.2 Model elements of VOD player 19
3.3 Input illustration 23
3.4 Combined input illustration 23
3.5 Input illustration revisited 24
3.6 Mobile Waiter trace matrix 27
3.7 Mobile Waiter trace matrix excert 32
3.8 Mobile Waiter trace matrix after analysis 33
3.9 Oracle questioning 36
3.10 Clause selector variables insertion 44
3.11 Mobile Waiter trace matrix after analysis 44

4.1 Clause selector variables on units level 47
4.2 Mobile Waiter trace matrix after analysis 47
4.3 Clause selector variables insertion 50
4.4 Mobile Waiter trace matrix after analysis 50
4.5 Mobile Waiter trace matrix after step 1 57
4.6 Mobile Waiter trace matrix after step 3 58
4.7 Mobile Waiter trace matrix after step 4 60
4.8 User guidance 61

5.1 Characteristics of assessed systems 63

Abbreviations

IR	**I**nformation **R**etrieval
XML	E**x**tensible **M**arkup **L**anguage
HTML	**H**yper**T**ext **M**arkup **L**anguage
xADL	E**x**tensible **A**rchitecture **D**escription **L**anguage
AG	**A**rtifact **G**roup
SAT	Boolean **Sat**isfiability
CNF	Bollean **Unsat**isfiability
MUS	**M**inimal **U**nsatisfiable **S**et
MSS	**M**aximal **S**atisfiable **S**et
MCS	**M**inimal **C**orrecting **S**et
HUMUS	**H**igh **L**evel **U**nion Of **M**inimal **U**nsatisfiable **S**ets

Chapter 1

Introduction

In this chapter we provide a short introduction to Software Traceability, subsequently we describe the goal of this thesis and its chapter structure.

1.1 Motivation

Nowadays software systems are tight connected to a large number of critical tasks in many areas of daily life and sometimes are able to replace human in many activities. The reliability of such systems must be very high in order to guarantee the people trust. It is important because failures in such systems may lead to major financial loss, damages or even loss of life. For example, passengers of an aircraft are sure that the flight control software works correctly and stable. Hospital personnel must be sure that patients remote monitoring system functions as intended.

Traceability is one of the prerequisites of the ability to attain the high level quality in the mentioned and similar software and thus the critical element of any rigorous system[1]. According The IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990) traceability is "the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another". A narrower definition of the traceability applied to the requirements traceability is the following: "Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through

its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)"[2].

In other words, the traceability describes a connection or relationship between artifacts within the development process. In the development process itself may be involved stakeholders with different interests and goals, so they will expect different types of traceability relations. For instance, end users may be interested in relationships between requirements and components of user interface, whereas testers may require relationships between requirements and test cases as well as between test cases and artifacts of source code.

The traceability can be successfully applied in areas like change impact analysis and change management, software validation verification and testing, software reuse, better artifact understanding [3], thereby increasing the quality and simplicity of software processes.

Approaches of the traceability capturing may be divided in manual, semi-automatic and automatic. There are also different approaches to storage of tracebility relations. In the present work we consider the *TM* (traceability matrix) as the method of traceability recording and representation. The TM is a document in the form of a table that correlates artifacts in respect to the traceability relations. In other words, each cell of the table contains information about traceability relation between the corresponding pair of artifacts.

One of the TM's disadvantages is that the relations stored in the matrix are typically captured manually. The manual discovering of traces is very error-prone and laborious process for large systems. The number of decisions to be made by an engineer is $m * n$, where n is the number of artifacts of a first type and m is the number of artifacts of a second type. On the other hand, it is important if not crucial for the system maintenance to have complete and accurate traces. However developers may not always have complete and precise knowledge about how requirements related to source code elements, that leads to uncertainties in input. One can distinguish two kinds of uncertainties[4]: partiality uncertainties and cluster uncertainties. If an engineer is not sure, whether some set of code elements implements the given requirement completely we talk about a partiality uncertainty. Cluster uncertainties arise, when an engineer understands the

role of groups of elements (e.g. classes), but there is a lack of understanding of purposes of individual members of the group(e.g. class methods).

To make the traces establishing process easier, developers or engineers working with traceability may provide *dependencies* between groups of artifacts instead of identifying trace relations between single artifacts directly[4, 5]. Dependencies serve as the foundation for automatic generation of trace relations. This process known as trace analysis can be performed using SAT-based reasoning, which allows to use the power of efficient SAT-solvers.

In this thesis we describe our vision of realization of the trace analysis using SAT-based reasoning.

1.2 Goals of this thesis

The goal of this thesis is to provide an approach and algorithms of trace analysis for the input presented by a set of dependencies. The approach must have the following key characteristics:

- It must support the traceability uncertainties[6] and be able to resolve them while trace analysis.
- It must support different types of traceability dependencies[4, 5] and perform the reasoning according the semantic of these dependencies, as well as fill up the trace matrix based on the reasoning results.
- The possibility of tolerating conflicts in the user input and performing analysis regardless of their presence with minimal to large systems with big trace matrices.
- The ability of the conflicts isolation (HUMUS strategy) and different isolation granularities to allow the level of details comfortable for an engineer.
- It must provide guidance to a user about possibly erroneous dependencies to help him/her to resolve conflicts as soon as possible.

Another important part of the thesis is to perform an exhaustive evaluation of the approach to assess:

- **Scalability**: approach must be applicable to large systems with big trace matrices.
- **Correctness**: algorithm must perform correct trace analysis according the semantic of dependencies.
- **Efficiency**: the method must show satisfactory results in the resolving of uncertainties and cover the maximal possible area of trace matrix with minimal possible information loss.
- **Isolation**: as long as the isolation is necessary to retain the automation in case of the presence of conflicts, the strategy must isolate as many erroneous dependencies as possible (recall) and as little correct dependencies as possible (precision).

1.3 Chapters description

The remainder of this thesis is structured as follows.

In the chapter 2 we provide an introduction to Software Traceability, describe different types of the traceability as well as methods and approaches of traceability capturing and traceability analysis.

In the chapter 3 we describe uncertainties which may occur while capturing traceability information and give the explanation of cluster and partiality uncertainties, methods for describing these uncertainties as well as approaches to trace analysis in case of the presence of uncertainties. This chapter also describes how we can apply SAT-based reasoning to the traceability analysis and provides an overview of conflicts resolution strategies.

The chapter 4 detailed all steps of our implementation of trace analysis algorithm using SAT-based reasoning, isolation strategies on units level and cells level, incremental reasoning, user guidance based on the incremental reasoning, and demonstrate the improvement of the oracle algorithm described in the chapter 3.

In the chapter 5 we discuss all steps and results of the evaluation we performed for the approach. We explain the algorithm we used for generating test cases for the evaluation, and provides information about configurations relevant for the assessment. We

demonstrate diagrams reflecting the results of testing and discuss the meaning of these results.

The chapter 6 is the introduction of TraceAnalyser - the tool supporting our method, its user interface and functionality as well as the description of Eclipse - the platform for our tool.

In Chapter 7 we present the conclusions we could draw from this work and discuss possible future extensions of the thesis.

Chapter 2

Traceability Basics

In this chapter we provide an introduction to Software Traceability, describe different types of the traceability as well as methods and approaches of traceability capturing.

2.1 Traceability relations

In order to provide meaningful interpretation of traceability, different types of traceability relations were proposed[3]. According Lindval and Sandahl[6] there are two general types of traceability relationships: *vertical* and *horizontal*. The former type can be applied if it is possible to establish dependencies between elements within a single model (e.g. between two elements of source code) and the latter one for describing a connection between artifacts of different models (e.g. requirements - source code). Spanoudakis and Zisman proposed in [3] the classification, which organizes different types of traceability relationships in eight groups:

- **Dependency relations** can be established between artifacts if their existence is interrelated or if changes of one artifact causes changes of another artifact. Dependency relations may be both vertical (e.g. between different requirements) and horizontal (e.g. between requirements and model elements).
- **Generalisation/Refinement relations** are used to describe how system components can be broken down in smaller elements or opposite how elements can be

combined to form other elements as well as how one elements can be refined by other elements.

- **Evolution relations** specify the evolution of artifacts. Under evolution in this case, one can understand the transformation of one element to another within system life cycle or replacement one element by another.

- **Satisfiability relations** can be established between artifacts if one of them meets expectations and needs of another one. A good example demonstrating the nature of satisfiability relations is relations between requirements and model elements. One can use them to ensure that the system fulfills requirements.

- **Overlap relations** are used to designate that artifacts involved in relations of this type refer to common features of the system. E.g. if some requirement is implemented by multiple code artifacts or described by multiple model elements it can be connected with these elements by overlap relations.

- **Conflict relations** are used to describe conflicts between different artifacts. Conflict relations can be established e.g. between requirements and model element to provide information about issues and about how these issues can be resolved.

- **Rationalisation relations** provide information about decisions made within the development process. They can e.g. describe why some element transforms in other element or prerequisites were a basis for creating model elements.

- **Contribution relations** are relations between a stakeholder and requirement artifacts contributed by the stakeholder. According [2] there are three main types of contributors: principals stimulating the creation are responsible for the influence of artifact on the system, authors who organize structure and content of the information in artifacts and documenters who capture and document this information.

Together with the classification of relation types it is also important to understand what types of artifacts may be connected by these relations. According Spanoudakis and Zisman[3] the following types of artifacts are the most significant: requirements, design artifacts, code artifacts and others(e.g. documentation, test cases etc.)

2.2 Purpose of the traceability

Before moving to more detailed description of traceability usage models and techniques of traceability capturing, we want to clarify which benefits one can gain from the traceability; in other words how useful is the traceability. Egyed et al.[7] showed the complexity and effort of capturing traceability information between requirements and code artifacts. They conducted an experiment, in which one hundred subjects recovered requirement-to-code traces (in methods and classes levels) for two open source projects (one half with industrial experience, another half - without). Their observations show that the effort of the traceability capturing increases with system complexity, however, more effort does not mean the better quality of traces. Granularity of artifacts has a direct impact on the effort (3-6 times more for the tracing requirements to methods than requirements to classes). Furthermore, authors mention that the automation plays a significant role while capturing traceability, but existing tools can not help to recover traces. As the process of recovering traces is laborious, it is vital to investigate, whether this process is justified in general and whether the use of traceability can significantly support the development and maintenance of software systems.

Mäder and Egyed conducted an experiment [8] in which 52 subjects were asked to perform 315 real maintenance tasks for two open source project. One half of the subjects could use traceability information, another half - not. The authors showed that the subjects with traceability information performed tasks on average 21% faster and created on 60% more correct solution than without it. The experiment shows that traceability can not only save time costs but can also increase the quality of the maintenance process.

Regan et al.[9] performed an analysis of implementation of traceability in real organizations and systematized their motivations for implementing traceability:

- **Regulation.** Many software development standards require the presence of the traceability implementation, especially for critical software like medical systems.
- **Safety case.** Safety critical systems must satisfy a number of non-functional requirements like reliability, security and availability. Moreover, such systems must be certified before entering service. Developers submit so called “safety case” as a proof that the system is safety and fulfills all such requirements. The safety case provides full traceability between requirements, code artifacts, test cases and risks.

- **Competitive advantage.** Traceability may help to reduce production costs and changes costs (through impact analysis) to a system. The latter property is especially valuable for the maintenance phase, where changed to any part of a system cause usually changes of other parts. So the traceability can help with more reasonable costs estimations and gives a competitive advantage.

- **Requirements validation.** Traceability may help to simplify the process of requirements validation. For instance, requirement-to-code links allow to check, whether the product fulfills all requirements. The opposite direction helps to identify the excess of functionality.

- **Rationale for decisions.** During Software System life cycle many key decision can be made. This is important if not crucial, to provide appropriate documentation of such decisions. They can be extremely useful for learning new team members, for system extensions, change management, maintenance etc. Traceability can support the documentation of decisions by providing traces from decisions to artifacts.

- **Change management.** If it is expected, that requirements will change frequently, traceability is a good way to support change management processes (e.g. the agile process). For instance, the traceability may trace requirements to the version and help to control, in which version some requirement is implemented.

Regan et al.[9] also mentioned that traceability can provide significant assistance in such activities as project management, risk management and defect management. Other motivators for applying the traceability are test coverage, easier program understanding, code maintenance etc. These motivators are relevant for both critical and common domains.

We listed above different types of traceability. It is also necessary to understand how useful each type can be and how it is applied in practice. Spanoudakis and Zisman provide in [3] various examples of applying different types of relations.

Knethen et al. consider in [10, 11] the usage of *dependency relations* between documentation artifacts (e.g. use cases, textual requirements) and logical artifacts (methods) that facilitate the impact analysis. An example of dependency relations mostly used

in practice is requirement-to-code relations. The authors also use the generalization relations to represent artifacts on different levels of abstractions.

Evolution relations can connect requirements in order to preserve the history of their changes or can be used to indicate how different artifacts (model elements, source code) are derived from requirements during the development process.

Overlap relations can be used for instance for connection of two or more documentation entities that describe the same logical artifact or for linking scenarios with other elements like use cases, code artifacts etc.

Conflict relations can be presented by inconsistency relations that connect requirements and design artifacts. They show that similar goals can not be achieved in two different specifications.

Let us consider more detailed how the traceability supports different development and maintenance activities. Spanoudakis and Zisman[3] provide also a comprehensive overview of fields where traceability can be successfully applied.

2.2.1 Traceability for change impact analysis and change management

This is one of the most significant areas, where traceability can be applied in order to determine the impact of changes in one part of the system to the another part and to figure out, whether such changes are necessary[3]. The simplest form of performing this analysis is the identifying of all entities of the system that can be affected by changes in some particular artifact. If the traceability information is available, this identifying can be relatively easy performed by organizing queries to retrieve traceability relation. This ability is supported by many traceability tools.

More complex impact analysis can be applied for classification of affected artifact in categories or for estimating effort or costs necessary for performing changes. Normally the retrieving of traceability relations between artifacts that are not directly connected is required in this case. It is obvious that the quality of both types of analysis depends on the quality, granularity, correctness, preciseness etc. of traceability relations. This fact has been also verified by a number of empirical studies[6].

2.2.2 Traceability for software validation, verification and testing

It is necessary to perform testing and other types of analysis, in order to ensure that a system under development meets all requirements and satisfy demands of all involved stakeholders. For this analysis traceability is able to provide a strong basis.

For example, contribution relations can be successfully used for identifying stakeholder and validation requirements with them. Dependency and satisfiability relations can support the analysis, whether all requirements are implemented in the model or source code artifacts. Similarly, the traceability relations can be used for identifying test cases related to requirements or other artifacts. Overlap dependencies can be used for automated validation of consistency of individual system components.

2.2.3 Traceability for software reuse

Researchers have acknowledged the potential of traceability relations in activities of identifying reusable artifacts in software development life cycle[6]. One can reuse not only elements of source code, but also model artifacts or requirements. In [12] dependency traces are applied for determining associations "requirement - model artifact - source code" for establishing so called application frames representing groups of artifacts of specific software applications which allow an engineer to determine reusable components within application frame, for example, by the analyzing of similarity of requirements in existing systems with requirements of the system to be developed. Another approach[11] suggests the use of existing requirement specification (or part of it) of families of systems for specification of requirements for a new family member that shares features with existing members. To identify the part of documentation that can be "recycled" overlap, one can use refinement and dependency relations.

2.2.4 Traceability for artifact understanding

A typical situation in the software product life cycle is where people involved in the maintenance are not contributors of artifacts, with which they work. It is especially important to understand the artifacts in the context of their creations as well as their references to other artifact (for instance, which requirements are implemented by this particular

code element, where should the engineer look for a code implementing this particular requirement or what documentation entities are related with this code). Traceability relations (e.g. dependency or overlap relations) in these cases are unavoidable. Rationalization relations can be used for providing an explanation of form of requirements or model artifacts.

2.3 Traceability retrieving, recording and maintenance

Mentioned areas are only examples, where traceability can be applied. However, even these examples show, how useful traceability may be. We assumed so far that traceability information is available and correct. This section describes methods of retrieving, recording and maintenance of traces.

2.3.1 Traceability retrieving

Spanoudakis and Zisman describe in [3] different approaches of traceability capture based on the level of automation for this process.

2.3.1.1 Manual creation of traceability relations

This group of approaches supposes manual declaration traceability relations between artifacts. However, there are different visualization tools supporting these activities (e.g. DOORS[1]) that provide convenient navigation through a set of artifacts to be traced. Despite of the tool support the effort of the manual establishing of traces may be high for large systems. Moreover, this approach supposes proper understanding of semantic of links to be established. Different stakeholder involved in the establishment process may have a different understanding of details that potentially leads to inconsistencies.

2.3.1.2 Semi-automatic generation of traceability relations

The group of this approaches can be again divided into two subgroups: *pre-defined link group* and *process-driven group*. The first group supposes generation of traceability

[1] http://www-142.ibm.com/software/products/us/en/ratidoor

information based on links previously predefined by users. The second group allows to capture traceability as a result of processes within development life-cycle.

An example of an approach from the pre-defined link group is proposed in [13]. It allows the collection of traceability information based on the observed scenarios of the software system and manual identification of hypothesised traces connecting these scenarios with artifacts. Using provided information new traces are generated between artifacts based on transitive reasoning (if A traces B and B traces C then A traces C).

An example of an approach from the process-driven group is PRO-ART[3]. This approach allows to generate traceability links as a result of creation and modification of artifacts in the development phase. To make it possible, all action in the used tools must be recorded and analyzed.

The mentioned approaches may be considered as improvement of manual approaches; however, they also have disadvantages. For example, links captured manually may still be error-prone, that leads to inaccurate results after automatic phase of generation traceability relations. In the case of process-driven approach results are dependent on used tools and the development methodology.

2.3.1.3 Automatic generation of traceability relations

Approaches from this group use different techniques in order to create traces automatically. Examples of these techniques are information retrieval (IR), traceability rules, inference axioms.

One of the approaches using IR described in [3]. Its idea is to use specific queries to determine traces between requirement documents and elements of source code. Such queries are constructed based on a list of identifiers extracted from source code artifacts. Next steps are the comparison of query with the set of requirements document, calculating similarity and ranking documents.

Another example is a method that generates traces between requirement statements, use cases and object models using XML-based traceability rules in order to identify syntactically related terms in the requirements and use case documents with the semantically related terms in an object model. Generated traces are represented as hyper-links.

In TOOR[14] traceability relations are captured by the usage of axioms. The tool allows to obtain traces between requirements, design and code artifacts.

2.3.2 Traceability recording

The next important question that should be covered in this chapter is how to represent and record traceability information. Spanoudakis and Zisman distinguish in [3] five different approaches of traceability recording.

2.3.2.1 Single centralized database approach

As follows from the name the approach supposes the usage a centralized database to store and maintain artifacts and traceability links between them. Examples of tools supporting this approach are DOORS[2] and TOOR [14]. The main benefit of this approach is that the post-processing of recorded traces may be performed based on effective queries to the database. On the other hand, it is not easy to work with relations between artifacts that were not originally created by the tool. In order to overcome this issue, some tools provide mechanisms of import. However, there are different limitation, which may restrict it(for example, import of artifacts created only by certain tools).

2.3.2.2 Software repositories

The main difference of software repositories from single databases is that the former provide a suitable flexibility in storing and manipulating and querying of software artifacts and traceability relations between them.

PRO-ART is the example of the requirements traceability environments that based on the software repositories approach [3]. It assumes integration of tools and specification of processes are used for manipulations with software artifacts on the top of the PRO-ART repository. Of course, such method requires additional effort to integrate and coordinate different tools, as well as model processes supported by these tools.

[2]http://www-142.ibm.com/software/products/us/en/ratidoor

2.3.2.3 The hypermedia approach

In order to avoid integration of a number of tools around a repository, some systems advocate an approach based on open hypermedia architectures. [3] describes a traceability manager *TraceM* that uses this approach. This prototype tool saves traceability relationships separate from the software artifacts. Relations are described as associations between artifacts by using metadata. The metadata specifies a type of artifacts that associated with relation, tools, in which these artifacts were created, transformers describing how to convert artifacts into the common TraceM format and integrators used for automatic identifying of trace relations. In order to get advantages of this service, developers must integrate tools for artifact creating with the TraceM environment by using standard techniques from open hypermedia environments.

2.3.2.4 The mark-up approach

This approach assumes the representation of traces separately from the artifacts using mark-up languages.

Gotel and Finkelstein [2] have developed a system to operate with contribution relations stored as hyperlinks using HTML. The rule based tool described in the section 2.3.1.3 uses XML to represent both artifacts and relation connecting the artifacts. The tool also uses translators to convert textual artifacts in XML-format. The advantage of this tool is that it does not require any integration of third-party software.

2.3.3 Traceability maintenance

Traceability once established must be maintained. Traceability maintenance means maintenance of traces between artifacts up to date if related artifacts are changed or removed, if new artifacts are added or if related traces are modified. Mäder and Gotel describe in [15] the importance of traceability: "*Without maintenance, traceability relations between elements get lost or represent false dependencies. Such a step by step degradation of traceability relations leads to traceability decay*". Authors provide in their work an exhaustive overview of the approaches of traceability maintenance, that briefly described below.

2.3.3.1 The event-based approaches

The approach provides a possibility to ensure that relations between artifacts are updated after modification of the artifacts. A system using this technique based on the event-notification mechanism has been developed by Cleland-Huang et al. that described in [16]. The purpose of the system is to record and maintain requirements-to-artifacts relations. Relations between a requirements and artifacts are stored in the system registry. If artifacts change, the system notifies all related requirements about changes.

2.3.3.2 Rule-based approaches

Murta et al. describe an example from this group in [17] and call this method *ArchTrace*. The approach allows to support evolution of traceability links between architectural models and implementation artifacts. ArchTrace uses *xADL* for a description of system architecture and *Subversion* for control of source code versions. Authors developed a number of policies that are triggered on committing a new version of artifacts. The policies are customizable and ensure the update of traceability relation in case of a new version of artifacts in a version control system.

2.3.3.3 Approaches based on recognizing evolution

Cleland-Huang et al. describe in [16] a method for identification of changes applied to requirements. Authors distinguish seven types of changes: create, inactivate, modify, merge, refine, decompose and replace. Building blocks for each change type are sequences of change actions: create requirement, set requirement attribute, create a link and set the link attribute. Authors provide an algorithm that allows to identify seven basic types of changes among sequence of captured change operations. Moreover, the identification mechanism itself should be triggered only for an entire user session in order to avoid false recognitions.

Chapter 3

Uncertainties And Conflicts In Traceability

In this chapter we describe uncertainties which may occur while capturing traceability information and give the explanation of cluster and partiality uncertainties, methods for describing these uncertainties as well as approaches to trace analysis in case of the presence of uncertainties. This chapter also describes how we can apply SAT-based reasoning to the traceability analysis and provides an overview of conflicts resolution strategies.

3.1 Uncertainties in the traceability

3.1.1 A Scenario-Driven Approach to Trace Dependency Analysis

For large systems manual discovering of traces between artifacts is error-prone and laborious process, since the number of decisions to be made by an engineer is $m * n$ if traces connect two perspectives with m and n artifacts. If one establishes relations between some architectural elements like requirements and, for example, elements of source code number of potential traces may be hundreds of thousands or even millions. Therefore, it is practically impossible for one engineer or for a group of engineers to establish all traces entirely correctly and completely. It is important if not crucial for the system maintenance to have complete and accurate traces. The goal of automated

TABLE 3.1: Test scenarios for VOD player

ID	**Scenario**
S1	Application shutdown and startup
S2	Set up applicaton
S3	Play stream video

approaches is to overcome the mentioned issues by minimizing or fully excluding manual intervention.

We briefly described the test-driven automated approach[13]. This section presents more detailed the foregoing approach and its extension as it is in a sense a departure point for the present work. The prerequisites of this approach are existing test scenarios for the system as well as its observable executable version. Additionally designer must provide few traces (so called *hypothesised traces*) linking development artifact (for example, requirements) with these test scenarios. They can be elicited from system documentation or corresponding models.

We use a simple illustration throughout this, which allows to understand the core principles of the approach(Figure 3.1).

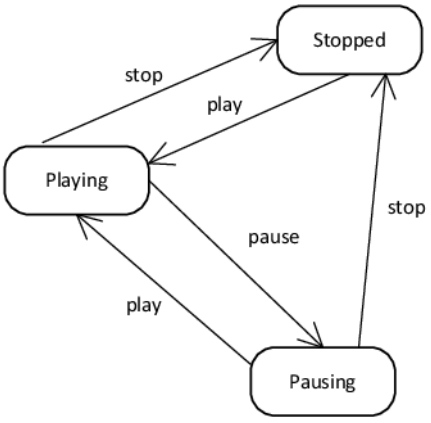

FIGURE 3.1: VOD Statechart diagram

The Figure 3.1 is an extract from the statechart diagram describing the behavior of VOD player[18]. Assume that we have three scenarios to be tested that are depicted in the Table 3.1, three model elements (states on statechart diagram) represented in the Table 3.2 as well as the set of elements of source code representing some classes of the system $\{A, B, C, D, E\}$.

Table 3.2: Model elements of VOD player

ID	Model Element
M1	State::Playing
M2	State::Stopped
M3	State::Pausing

As input for the approach one uses traces between scenarios and code as well as between scenarios and model elements. Both types of traces may be provided by designers. The latter may be also derived automatically during testing of scenarios. The result of the approach are traces between model and code elements. Selected scenarios are executed on the software system. It is important to be able to observe internal activities of the system while its execution. Such ability may be provided by external monitoring tools (e.g. Rational PureCoverage®from Rational Software). Elements of source code (for instance, classes, methods, lines of code) that used while executing form footprint of execution. Footprints of all scenarios are captured in the form of a graph in order to identify overlaps between them. For example, if two scenarios *S1* and *S2* use the same lines of code then the approach may determine a dependency between these scenarios. If initial traces were determined between *S1* and the code element *A* as well as between *S2* and *B*, one can derive a dependency between *A* and *B*. This can be concluded in the ***commonality principle***: *if S1 traces to some code A and S2 traces to some code B, then a trace dependency exists if A and B overlap.*

This approach is useful for both identification of new traces and validation of provided *hypothesised traces*, so far as conflicts in the footprint graph point conflicts in the existing traces. The approach is also able to identify ambiguities like *S1* traces to *B* or *C* or both. One of the advantages of this approach is the possibility to identify traces among any model elements that somehow related to source code regardless their nature (methods in class diagrams, states of statechart diagrams). Initial hypothesised traces should be normally established manually; however, this process becomes more automated. For instance traces derived by this approach may also be used as hypothesised traces.

One of the serious faced challenges of this approach is the ambiguity. The following examples demonstrate what types of ambiguity may arise:

- *M1*, *M3* trace to the code of scenario *S1*, but it is uncertain whether they also trace to a code other than the code of the scenario *S1*.

- It is uncertain whether *M2* relates also to the scenario *S1*.
- It is uncertain what part of the observed code of scenario *S1* used by *M1* and *M1*.

Allowing ambiguity is very powerful mechanism because if developers only have partial knowledge on some system components it allow to express dependencies to the level of details, with which engineer are comfortable. Therefore, the expression of hypothesised traces through inclusion and exclusion was proposed. Inclusion defines those model elements that relate to the scenarios. Trace from the scenario *S1* to the model elements *M1* is the example of inclusion. Opposite, the exclusion describes that some model elements is certainly not related to some scenario (for example, *S1* is not scenario for class *A*).

Egyed accentuates in [4] benefits of this approach as the following:

- To infer n^2 trace dependencies (potentially every model element to every other) only n input hypotheses are needed.
- Semantic and syntactic differences between model elements are irrelevant. It is not necessary to understand the difference between any two model elements (n^2 differences). Instead, it is only sufficient to understand the difference in meaning between a model element, its test case, and the system (n differences)
- Developers only need to investigate their model elements and how they related to the system without complete understanding other elements and communications with other developers.
- It is possible to use informal or partial notations.

The trace analysis can be performed straight forward if precise footprint is known for each model element via reasoning: $(m \rightarrow s) \wedge (s \rightarrow f) \Rightarrow (m \rightarrow f)$. In other words, a model element m has a footprint f if m is related to a test scenario s and this test scenario has a footprint f. We assume the footprint f of execution of a test scenario s to be correct because it was captured automatically while execution of scenarios. Opposite, test scenarios are associated with model elements manually and errors in the associations can not be excluded. This may lead to the problem when the execution results in a subset of the real footprint if test cases do not cover the entire scope of model element.

Another problem appears if engineers are not aware of all model elements related to some scenario. In this case related model elements will have a larger footprint than the real one.

The presented approach is not able to handle the mentioned problems. Another its weakness is the impossibility to identify *shared code* executed in multiple test scenarios but not related to any of them (general purpose and domain-independent code). To overcome these issues, Egyed proposes an extension of the approach[4].

3.1.2 A Scenario-Driven Approach to Trace Dependency Analysis: Extention

If an engineer would have precise knowledge in some set of models elements, exact footprint f could be derived for some model element m from this set. In this case we know that m traces only to f and f belongs only to m. However, developers may not always have complete and precise knowledge about how model elements are related to code elements. This may lead to *uncertainties* in input. In his work[4] Egyed differs two types of uncertainties:

- *Partiality uncertainty* occurs if it is not known fully was something is. For example if an engineer is not sure whether all tests were performed to form the complete and correct footprint for some element. In this case one can say that the model element is at least the given footprint. If an engineer is not sure about whether a given model element entirely captures given scenario, we can say that the model element is at most the given footprint. Engineer may also want to express that there is no relation between some model is certainly not related to a footprint. This can be expressed through *is not* relation.

- *Cluster uncertainty* determines situation where roles of individual elements in a group is not known. For instance developers may not known the role of some methods in class but they are sure about the role of this class as a whole. This principle is also applicable to model elements, for example, if it is easier to state that model elements m_1 and m_2 refer together to the footprints f_1 and f_2 than to identify individual relations.

Both types of uncertainties give engineers the ability to determine relations to the level of detail, with which they are comfortable. The good point is that it is still possible to perform trace analysis with the presence of uncertainties. The extended approach allows also to identify and isolate shared code. So if two model elements overlap by the usage of shared code this should not lead to trace between them.

Now we consider the revisited approach detailed. Trace analysis has to perform two actions. First of them is to identify for each code artifact model elements, to which it belongs. If the code artifact belongs to some model element, this model element is *included*. If the code artifact does not belong to the model element, the model elements is *excluded*. If we are uncertain about this fact, the model element is neither *included* nor *excluded*. The second action to be performed is to identify shared code that has to be ignored during trace analysis. Trace analysis is complete if every code artifact includes or excludes every model element or if this code artifact is shared.

To support cluster uncertainties developers may combine elements. Thus, the input describing relationships between model elements and code elements may be provided by groups of elements or over individual elements. So the input describing that m_1 and m_2 refer together to the footprints f_1 and f_2 may be designated as $(m \rightarrow f)$, where $m = \{m_1, m_2\}$ and $f = \{f_1, f_2\}$. Note that this input does not depict individual relations but rather relations among groups of elements. To support partiality uncertainties developer may differ types of such relations. We designated $(m \rightarrow f)$ relation as ***is***, that means $m\ is\ f$. The ***is*** relation may be qualified with the following types: “isAtLeast”, “isAtMost”, “isExactly” and “isNot”.

The overlapping of footprints f_1 and f_2 is a normal situation. In this case we can separate the source code represented by f_1 and f_2 into fragments indicating overlapping and non-overlapping parts. This fragments we call *code elements* , and we can describe input through these code elements. If some input refers to some code elements, it refers to *all* lines of code within the code elements. If an input does not refer to code elements, it does not refer at all to any lines within the code element. We demonstrate this on an example. Consider three model elements m_1, m_2 and m_3 and three code elements c_1, c_2, and c_3. Assume we have the following input visually represented in Table 3.3:

- $\{m_1, m_2\}\ isExactly\ \{c_1, c_2\}$

  In this case m_1 and m_2 are included (I) in both c_1 and c_2, but excluded (E) from

TABLE 3.3: Input illustration

		c_1	c_2	c_3
$\{m_1, m_2\}\ isExactly\ \{c_1, c_2\}$	m_1	I	I	E
	m_2	I	I	E
	m_3			

		c_1	c_2	c_3
$\{m_2, m_3\}\ isExactly\ \{c_2, c_3\}$	m_1			
	m_2	E	I	I
	m_3	E	I	I

TABLE 3.4: Combined input illustration

		c_1	c_2	c_3
Both	m_1	I	I	E
	m_2	E	I	E
	m_3	E	I	I

c_3

- $\{m_2, m_3\}\ isExactly\ \{c_2, c_3\}$

  In this case m_2 and m_3 are included (I) in both c_2 and c_3, but excluded (E) from c_1

These two inputs provide different sets of included and excluded elements and should be combined (Table 3.4). m_1 is excluded from c_3 because of the first input. m_2 is excluded from the c_1 (second input) and c_3 (first input). m_3 is excluded from c_1 (second input).

Trace analysis allows to identify similarities among model elements that belong to some perspective. An example of such perspective may be a class diagram. Every class in the class diagram describes some specific part of the system and, therefore, has some specific code not shared with any other class. Other examples are state chart diagram, object diagram, etc. If we consider m_1, m_2, m_3 as elements of the same perspective the input implies that c_1 and c_2 must be unique part for m_1 and m_2. Therefore, the unique code for m_3 is outside c_1 and c_2. In the considered example, the only possibility left is c_3. So we conclude that m_3 relates to some subset of c_3. We talk about the subset because it is unknown, whether the given perspective describes the system entirely. Moreover, some subset of the footprint c_1 and c_2 is unique for m_1, other subset is unique for m_2. Remaining subset if not empty represents a shared code. So the logical consequence for the example is: footprint c_1 and c_2 are shared for all model elements other than m_1 and m_2, and so it is shared for m_3. Taking these logical consequences into account we can

TABLE 3.5: Input illustration revisited

		c_1	c_2	c_3
$\{m_1, m_2\}$ *isExactly* $\{c_1, c_2\}$	m_1	I	I	E
	m_2	I	I	E
	m_3	∘	∘	I

		c_1	c_2	c_3
$\{m_2, m_3\}$ *isExactly* $\{c_2, c_3\}$	m_1	I	∘	∘
	m_2	E	I	I
	m_3	E	I	I

		c_1	c_2	c_3
Both	m_1	I	∘	E
	m_2	E	I	E
	m_3	E	∘	I

revisit the illustration in Table 3.3 and Table 3.4(Table 3.5 ; shared code designated as ∘)

$(m \rightarrow f)$ defines that m traces to f and f includes m. However, there are uncertainties in the input because it is unknown whether m traces to footprints other than f (the footprints other than f we denote as $F - f$), or other model elements $(M - m)$ do trace to the footprint f. To express the uncertainty the following types of "is" relations were defined:

- m *isAtLeast* f. M has at least footprint f (minimal footprint for m), but the real footprint may be larger.
- m *isAtMost* f determines the maximal possible footprint for m and implies that m is excluded from all footprints $F - f$. This type of relation expresses also the uncertainty that the real footprint of m may be less than f.
- m *isNot* f expresses that m is excluded from f and, therefore, must be included in some footprint $F - f$. With such properties *isNot* relation is nothing else, but the negation of *isAtMost* relation.
- m *isExactly* f relation determines that f is the exact footprint for m, and, therefore, cannot be less or larger than f. The direct consequence is that m is excluded from every footprint in $F - f$.

One uses for the trace analysis specific data structure called *footprint graph*. Each node contains a code element or set of code elements and list of excluded, included and shared model elements for the set of code elements. An edge defines a parent-child relationship between two nodes where the child has a subset of the code elements of the parent. If two model elements overlap on some node of the graph, we derive trace between them. Different model elements of different perspectives will rarely have the same footprint. So we can use the common code for model elements for measuring the similarity. The more common code have two model elements the more similar they are. The output may still contain uncertainties if some nodes remain incomplete.

3.1.3 Traceability uncertainties between architectural models and code (Related work)

One uses the mentioned *is-relations* to express dependencies between model elements and footprints in order to determine traces between model elements. These relationships may be also extended in order to describe direct trace relations between model elements and elements of source code. Ghabi and Egyed proposed such extension in [5] that transforms is-relations to *implements-relations*. We call these relations in the present work *dependencies*. However, this transformation does not contradict with the original semantic of is-relation. One can consider it as a specializing of the common model-to-model relation to the model-to-code relation. Similarly to the original language, the extension allows to express incompleteness and uncertainties (for instance, an engineer knows that some code element implements some requirements, but the engineer is not sure, whether this code element does also implement other requirements or whether this requirement is also implemented by another code elements). As long from one side we always deal with code elements, the language can describe again two kinds of uncertainties:

- *Cluster uncertainty.* Developers may understand the role of groups of elements (for example, some class) good but may not know the role of single group members (lile methods of the class).
- *Partiality uncertainty.* Developers may be not sure, whether some set of code elements implements the given model element entirely.

We are interested in two types of relations between individual elements: $trace(m, c)$ expressing that model element m is implemented by the code element c or that c implements m, and $no-trace(m, c)$ expressing precise knowledge, that c does not implement m. These two kinds of relations, if derived, allow us to build trace matrix from the input.

As long as we deal with cluster and partial uncertainties, it is necessary to be able to express relationships among groups of elements. For this purpose Ghabi and Egyed introduced in [5] objects called *artifact groups* (AG). Artifact groups link code element with a set of model elements or vice versa. For example, $AG(c, \{m_1, m_2, ..., m_k\})$ expresses that code element c implements some subset of $\{m_1, m_2, ..., m_k\}$ of model elements but at least one of them. $AG(m, \{c_1, c_2, c_3, ..., c_k\})$ expresses that model element m is implemented by some subset of code elements but again by at least one of them. If $\{c_1, c_2, c_3, ..., c_k\}$ is a set of all methods of some class, then such artifact groups indicate that m is implemented by this class, what means by at least one of the class methods, but it may be unknown, which methods exactly are involved.

The goal of the reasoning now is to retrieve maximum information from the user input to derive $trace$ and $no-trace$ relations between individual elements using logical consequences of the input. The approach described in the paper provides a mechanism for it.

For illustration we will use a part of the trace matrix (Table 3.6) for Mobile Waiter application for Android platform. The purpose of this application is to support activities of waiter in restaurants. Trace matrix presented in Table 3.6 identifies which classes implement which requirements. In order to make the illustration more readable we will use the abbreviations of requirements:

- *CrO*(Create Order): The application must provide end user the possibility to create a new order.
- *ClO*(Close order): The application must provide end user the possibility to close existing orders.
- *CSR*(Create Separate Receipt): The application must provide end user the possibility to create separate receipts for one order.

Table 3.6: Mobile Waiter trace matrix

	01: CrO	02: ClO	03: CSR	04: ChP	05: BgU
CalculatorActivity			×		
MainActivity	×			×	
OrderActivity	×	×	×		
PreferencesActivity				×	
Values	×	×	×	×	×
action.Action	×				×
action.Request					×
data.OrdersData	×	×	×		
data.Offline				×	
data.Utils	×	×	×		×
models.Calculator			×		
models.Order	×	×	×		×
models.Table	×	×			×
ui.Factory	×		×		
ui.UIDefaultElement	×		×		
ui.UITable	×	×			
ui.UIOrderPositionElement	×		×		
ui.UIProperties	×		×		

- *ChP*(Change Preferences): The application must provide end user the possibility to change certain preferences.
- *BgU*(Background update): Thepplication must perform synchronization with the server on background without interrupting the user's work.

For simplicity, we will use only five classes:

- `Values`. This class is the collection of common global values, used by the application.
- `Action`. This class is a collection of basic actions performed by application through the life cycle like executing background updates, sending requests to the server, parsing answers from server, etc.
- `Request` is used to send requests in the background to the server.
- `Utils` contains common functions on data that are used within the application.
- `Order` encapsulates all data related to orders and methods for creating, modification and closing orders.

As model elements we use requirements and denote a set of all requirements as R. Code elements are classes that build a set C. $\{r^*\}$ and $\{c^*\}$ denote some subsets of R and C respectively.

The first step is to derive *logical units* describing uncertainties (AGs) as well as certainties ($trace$, $no-trace$) from dependencies based on their semantics. We distinguish four types of dependencies coinciding with types of $is-relations$.

ImplAtLeast dependency $\{r^*\}$ $ImplAtLeast$ $\{c^*\}$ expresses that requirements from $\{r^*\}$ are implemented by all of the code elements from $\{c^*\}$ and possibly more. In other words, every element of $\{c^*\}$ must implement at least one requirements from $\{r^*\}$. Following artifact groups are derived:
`for each` r `from` $\{r^*\}$`:` `derive` $AG(r, \{c^*\})$;
`for each` c `from` $\{c^*\}$`:` `derive` $AG(c, \{r^*\})$;

For example, let us consider the following dependency:

`Dependency 1:` $\{$`ClO, CrO`$\}$ `ImplAtLeast` $\{$`Values, Action`$\}$.
So we derive the artifact groups:

```
AG(ClO, {Values, Action})
AG(CrO, {Values, Action})
AG(Values, { ClO, CrO })
AG(Action, { ClO, CrO })
```

$AG(ClO, \{Values, Action\})$ describes that the requirement ClO is implemented by the class $Values$ or $Action$ or both of them. At the same time the artifact group does not provide any information, whether ClO is implemented by other classes or not.

$AG(Values, \{ClO, CrO\})$ expresses, that the class Values implements the requirement ClO, or the requirement CrO or both of them. Similarly $AG(Values, \{ClO, CrO\})$ does not restrict relations between the class $Values$ and other requirements.

ImplAtMost dependency $\{r^*\}ImplAtMost\{c^*\}$ expresses, that requirements from $\{r^*\}$ are implemented by some of the code elements from $\{c^*\}$ and certainly not more. ImplAtMost dependency does not imply that every code elements from $\{c^*\}$ must necessarily implement at least one of the requirements $\{r^*\}$ but determines, that $\{r^*\}$ can not be implemented by code elements not from $\{c^*\}$. Following artifact groups are derived:
`for each` r `from` $\{r^*\}$`:` `derive` $AG(r, \{c^*\})$;
`for each` c `from C-`$\{c^*\}$`, r from` $\{r^*\}$`:` `derive no-trace(c, r);`

For example, let us consider the following dependency:

`Dependency 2:` $\{ChP\}$ $ImplAtMost$ $\{Request, Action\}$.
So we derive the artifact groups:

```
AG(ChP, {Request, Action})
no-trace(ChP, Values)
no-trace(ChP, Utils)
no-trace(ChP, Order)
```

No trace units build relations between individual artifacts describing that the requirement *ChP* is not implemented by any of the classes $Values$, $Utils$ and $Order$.

ImplExactly dependency $\{r^*\}$ $ImplAtExactly$ $\{c^*\}$ expresses, that each requirement from $\{r^*\}$ is for sure implemented by some subset of code elements form $\{c^*\}$, each code element from $\{c^*\}$ implements at least one requirement from $\{r^*\}$, and no requirements from $\{r^*\}$ are implemented by any code elements other than from $\{c^*\}$. However, code elements from $\{c^*\}$ may still implement requirements not from $\{r^*\}$. Following artifact groups are derived:
`for each` r `from` $\{r^*\}$: `derive` $AG(r, \{c^*\})$;
`for each` c `from` $\{c^*\}$: `derive` $AG(c, \{r^*\})$;
`for each` c `from C-`$\{c^*\}$`, r from` $\{r^*\}$: `derive no-trace(c, r);`

For example, let us consider the following dependency:

`Dependency 3:` $\{ClO\}$ $ImplExactly$ $\{Values, Utils, Order\}$.
So we derive the artifact groups:

```
AG(ClO, {Values, Utils, Order})
AG(Values, {ClO})
AG(Utils, {ClO})
AG(Order, {ClO})
no-trace(ClO, Action)
no-trace(ClO, Request)
```

ImplNot dependency $\{r^*\}$ $ImplNot$ $\{c^*\}$ determines that every requirement from $\{r^*\}$ is not implemented by any code element from $\{c^*\}$. This indirectly implies that every requirement must be implemented by remaining code artifacts from $C-\{c^*\}$. However, we do not accept this for a fact without explicit input from the engineer. That means that ImplNot dependency does not produce any artifact groups, but only no-trace relations between each pair of artifacts:
`for each` c `from` $\{c^*\}$: `derive` $no-trace(r, c)$;

For example:

`Dependency 4:` $\{CSR\}$ $ImplNot$ $\{Action, Request\}$.
So we derive the no-trace relations:

```
no-trace(CSR, Action)
no-trace(CSR, Request)
```

The common actions of generating artifact groups and no-trace relations from the input dependencies are showed in the algorithm 1.

Algorithm 1 Deriving artifact groups and no-trace relations

```
1: U ←< EMPTY >                                   ▷ initialize empty set
2: for all {r_1, r_2, ...} ImplAtLeast {c_1, c_2, ...} do
3:     for all c_j do
4:         U ← U ∪ AG(r_i, {c_1, c_2, ...})
5:     end for
6:     for all r_i do
7:         U ← U ∪ AG(c_j, {r_1, r_2, ...})
8:     end for
9: end for
```

Analogously to the approach described in [4] all input data is collected in the structure called *the footprint graph*. However, the footprint graph is different and contains nodes for each code elements (*CE-nodes*) and each model element (in our case for each requirement, *RE-nodes*). *CE-nodes* may be connected to *RE-nodes* by two types of edges: edges of no-trace relations (depicted as dashed lines) and edges of trace relations (depicted as solid lines). Note, that we can not derive trace relations directly from the input as we do for no-trace relations. Trace relations and additional no-trace relations may be obtained as a result of logical reasoning based on the input data. The reasoning process we describe detailed below. The footprint graph is also able to capture uncertainties in input, described as artifact groups. Each artifact group is represented through two connected nodes. One of the nodes is either a CE-node or a RE-node; the second one represents set of elements in some artifact group. Edges between these two nodes are depicted as thick solid line and points uncertainty.

Figure 3.2 depicts the footprint graph constructed for four example dependencies discussed above. The two node columns in the middle describe requirement artifacts and code artifacts. Dashed lines between them reflect no-trace relations derived from input. Left and right columns contain nodes for describing artifact groups. They connected with thick lines to the requirement nodes or code element nodes. Such pairs reflect artifact groups. E.g the top left node $\{Action, Values\}$ connected with the RE-node *CrO* describes the artifact group AG(CrO, {Values,Action}) from the dependency 1. The right down node $\{ClO\}$ connected with the CE-node *Order* describes the artifact group AG(Order, {ClO}) from the dependency 3.

Dependencies and derived artifact groups and no-trace relations are the foundation for automatic trace generation. Dependencies provide the freedom to work with artifacts to an arbitrary level

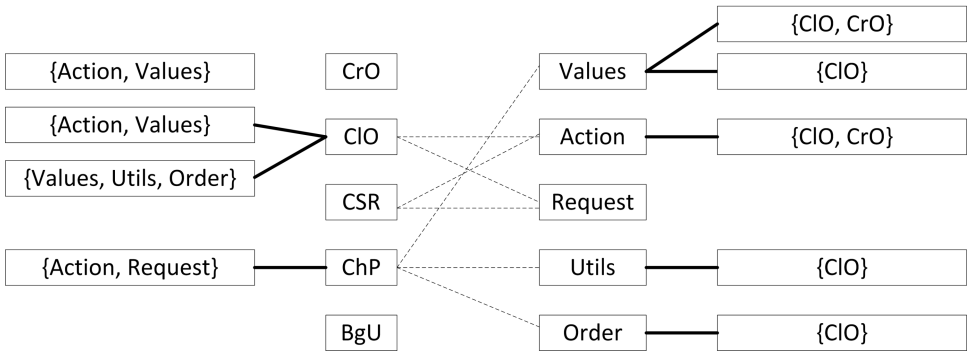

FIGURE 3.2: Footprint graph

of details. However, except initial no-trace relations, only relations between groups of artifacts are produced. More valuable is to derive from the provided input relations between individual artifacts (each pair requirement-code artifact). We do not consider in the present work relations between two elements of source code or two requirements. The approach proposed by Ghabi and Egyed in [5] uses propagation rules and correctness constraints to derive individual trace relations between pairs of artifacts:

- **Propagation rules for reducing uncertainties.** Consider the system described by the matrix in Table 3.6 again. The Dependency 1 results in the artifact group AG(ClO, $\{Values, Action\}$) (ClO is implemented by $Values$ or $Action$ or both). At the same time, the Dependency 3 results in the no-trace relation no-trace(ClO, $Action$) expressing that ClO is not implemented by $Action$. Based on this we can reduce the artifact group AG(ClO, $\{Values, Action\}$) to AG(ClO, $\{Values\}$). By analyzing all no-trace relations initially derived from the input one can reduce uncertainties in the artifact groups by removing requirements or code artifacts according the following rule:

```
for each no-trace (r, c)
        for each AG(r, {c_1, c_2, ...})
                if(c_i == c))
                        remove c_i from AG(r, {c_1, c_2, ...})
        for each AG(c, {r_1, r_2, ...})
                if(r_i == r))
                        remove r_i from AG(c, {r_1, r_2, ...})
```

- **Propagation rules for suggesting trace.** The Example above demonstrates the reduction of an artifact group to the group with single artifact AG(ClO, Values). According the

Table 3.7: Mobile Waiter trace matrix excert

	01: CrO	02: ClO
OrdersData(OD)		
Offline(OL)		
Calculator(CA)		

semantic of artifact groups one can conclude, that the requirement ClO must be implemented by the code artifact Values. Obviously we can extend this for all artifact groups with single artifact:

```
for each AG(artifact, {setOfArtifacts})
    if(setOfArtifacts.size==1)
        derive trace(artifact, setOfArtifacts[0])
```

Reducing uncertainties and suggesting trace allow to identify traceability relations between individual artifacts, but if we apply these rules "as is" conflicts may occur. We will demonstrate such conflict on an example. We keep all four dependencies and add the new one: *Dependency 5:* {*ClO*} *ImplNot* {*Order*}.

After the reduction we obtain an artifact group *AG(Order, {})*. The original state of this artifact group derived from the Dependency 3 was *AG(Order, {ClO})*. We remove the artifact *ClO* from this group by applying the propagation rules to the relation $no - trace(ClO, Order)$, derived from the dependency 5. Artifact groups with empty artifacts set signal about the conflict. They may be obtained only from with exactly one artifact if there is a corresponding no-trace relation. From the other hand, we derive trace relation from artifact groups with single artifacts. In this case both trace and no-trace relations are established between the same pair of artifacts. This is the conflict.

In the chapter 4 we will show, how to handle and support conflicts, but now we assume that the input is consistent and error free. This is guaranteed by correctness constraints:

- Every artifact group must have at least one target artifact. Otherwise we observe a conflict like the described above.
- One cannot establish trace and no-trace relations between a pair requirement and code element at the same time. This rule is in principle the implication from the first one.

We demonstrate the process with an example. Consider an excerpt of the trace matrix for the Mobile Waiter Application to be filled as illustrated in the Table 3.7.

TABLE 3.8: Mobile Waiter trace matrix after analysis

	01: CrO	02: ClO
OrdersData(OD)	T	
Offline(OL)	N	
Calculator(CA)	N	

We need to fill up the matrix based on the following input:

`Dependency 1:`$\{CrO\}\ ImplExactly\ \{OD\}$

`Dependency 2:`$\{CrO\}\ ImplAtMost\ \{OD, OL\}$

From the first dependency based on the semantic of *ImplExactly* we generate a set of artifact groups and no-trace relations:

```
AG(CrO, {OD})
AG(OD, {CrO})
no-trace(CrO, OL)
no-trace(CrO, CA)
```

From the second dependency based on the semantic of *ImplAtMost* we produce the next set of artifact groups and no-trace relations:

```
AG(CrO, {OD, OL})
no-trace(CrO, CA)
```

The next step is reducing uncertainties. We apply the rule to each no-trace relations. E.g. *no-trace(CrO, OL)* reduces the artifact group *AG(CrO, {OD, OL})* to *AG(CrO, {OD})*. Other no-trace relations have no effect for this input. Finally each artifact group with single target artifact generates a trace relation (Propagation rules for suggesting trace):

```
AG(CrO, {OD})       →  trace(CrO, OD)
AG(OD, {CrO})       →  trace(CrO, OD)
no-trace(CrO, OL)
no-trace(CrO, CA)
AG(CrO, {OD})       →  trace(CrO, OD)
no-trace(CrO, CA)
```

Table 3.8 demonstrates the resulting trace matrix after analysis. T in cell corresponds to the trace relation. N in cell corresponds to the no-trace relation. Empty cell indicates that no information about the relation for this pair of artifacts is provided or can be extracted from the input. As no correctness constraints were violated during analysis, the input is consistent. We allow the cell to be empty not only if there is no information provided about it, but also in the cases if the analysis does not provide the unambiguous information about the cell. For instance, if after applying of all propagation rules we have an artifact group like $AG(c, \{r_1, r_2\})$ and no artifact groups like $AG(c, \{r_1\})$ and $AG(c, \{r_2\})$, we can only conclude that at least

one of the requirements r_1 and r_2 are implemented by the code c, but we can not say exactly which requirement r_1 or r_2 is implemented by the code c without additional input. Matrix cells $c - r_1$ and $c - r_2$ may contain both T and N (cell contains TN) that gives in principle not more information than empty cell caused by missing input. So we assume empty cells and TN-cells equal and will use empty cells for both cases.

3.2 SAT-based reasoning for the tracebility analysis

The method for deriving trace relations based on propagation rules described above shows good results for problems of relatively small size. For cases with more than 500K artifact groups the approach requires much more time, what restricts the scalability. We keep the original semantic of all language elements (relations, artifact groups, dependencies) as well as the idea, how to reduce uncertainties but propose another algorithm using SAT-based reasoning that allows to increase the scalability.

Next is a brief introduction to Boolean satisfiability(SAT) problems terminology. Boolean satisfiability problem can be short described as following: establish for a given Boolean formula, whether variables may be assigned to make the formula evaluate to *true*. If no such assignment exists we say that the formula is *unsatisfiable*, otherwise it is *satisfiable*. SAT problems are defined in conjunctive normal form (*CNF*). A formula in conjunctive normal form is a conjunction of *clauses*. Each clause is a disjunction of *literals*. A literal is a variable or a negation of a variable. Each variable may be equal to *true* or *false*. Assumptions are assignments for literals that constrain the assignment possibility of a literal to either true or false. All these elements are depicted on Figure 3.3.

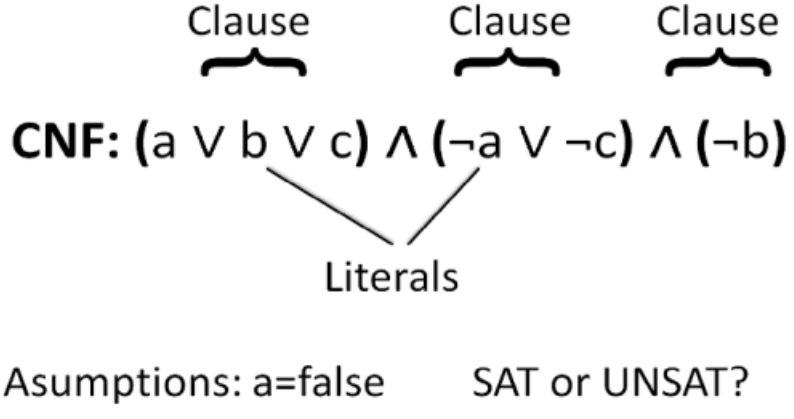

FIGURE 3.3: A CNF example

For example, one can see that the CNF $(a \vee b \vee c) \wedge (\neg a \vee \neg c) \wedge (\neg b)$ is satisfiable because the assignment $a = true$, $b = false$, $c = false$ evaluates the CNF to true.

SAT Solver is an algorithm that determines for a given CNF instance, whether it is satisfiable or not. *Complete SAT* solver always finds an assignment if the problem is satisfiable. SAT problem is NP-complete, but modern algorithms typically use sophisticated methods to work faster and can solve large problems for an appropriate time.

As there are efficient SAT-solvers, we can use their potential for the trace analysis. For this purpose we need to encode the input provided by an engineer as CNF in the way that each variable would represent some cell of the trace matrix, in other words would describe a relation between a pair of artifacts. As we are interested in two types of such relations (trace, no-trace), it is sufficient to encode relation for each pair of artifacts as a Boolean variable. Equality of a variable to $false$ will designate the no-trace relation between the pair of artifact, whereas $true$ value will correspond the trace relation. The maximal number of variables in CNF is equal to the number of cells in the trace matrix but may be also less. Clear, if no input provided about some cell, we do not need to include corresponding variable in CNF, it suffice to assume this cell to be empty.

We use dependencies, generated no-trace relations and artifact groups as the contributors of clauses of the CNF.

The meaning of the no-trace relation $no-trace(r,c)$ is that requirements r is not implemented by the code artifact c. The cell $r-c$ must contain N in this case and the Boolean variable for this cell t_{rc} must be equal to $false$. Absolutely the same semantic has the clause $(\neg t_{rc})$ that forces the variable t_{rc} to be $false$, otherwise the entire CNF becomes unsatisfiable. Therefore, the translation of no-trace relations to clauses is straight forward.

The meaning of the artifact group $AG(r,\{c_1,c_2,\ldots,c_k\})$ is that the requirement r is implemented by some subset of $\{c_1,c_2,\ldots,c_k\}$ of code elements but at least by one of them. That means that at least one of the cells $r-c_1,\ldots,r-c_k$ must not contain N (but may be empty), what implies in one's turn the inadmissibility of equality of all variables $t_{rc1},\ldots,t_{rck}$ to $false$ simultaneously. This property is independent on types of source and target artifacts of the artifact group and can be also applied to groups like $AG(c,\{r_1,r_2,...,r_k\})$, so the variables $t_{r1c},t_{r2c},\ldots t_{rkc}$ cannot be equal to $false$ simultaneously. Exactly the same restriction and, what is valuable, the same semantic gives the clause $(t_{rc1}\vee\ldots\vee t_{rck})$. It expresses that at least one of the containing variables must be equal to $true$, but variables from an arbitrary subset (except empty set) may simultaneously be equal to $true$. The translation of artifact groups to clauses is also straight forward.

This basis allows us to translate all types of dependencies into clauses. The result of the translation process is the set of clauses, whose conjunctions is a CNF. Below we demonstrate such trunslation process for two example dependencies:

$\{r_1\}$ *ImplExactly* $\{c_1\}$

AG(r_1, $\{c_1\}$)

AG(c_1, $\{r_1\}$) $\rightarrow$ (t_{r1c1})

no-trace(r_1, c_2) $\rightarrow$ $(\neg t_{r1c2})$

no-trace(r_1, c_3) $\rightarrow$ $(\neg t_{r1c3})$

$\{r_1\}$ *ImplAtMost* $\{c_1, c_2\}$

AG(c_1, $\{c_1, c_2\}$) $\rightarrow$ $(t_{r1c1} \vee t_{r1c2})$

no-trace(r_1, c_3) $\rightarrow$ $(\neg t_{r1c3})$

Resulting CNF as a conjunction of clauses takes the form $(t_{r1c1}) \wedge (\neg t_{r1c2}) \wedge (\neg t_{r1c3}) \wedge (t_{r1c1} \vee t_{r1c2}) \wedge (\neg t_{r1c3})$

The analysis of the CNF shows, that it is satisfiable, so one can conclude, that the input is consistent. However, this information is not sufficient to identify the values of individual cells (values of each variable). To do this, we can use the SAT solver as *an oracle* giving the allowed values for each cell. The oracle is based on the ability of SAT solver to check the satisfiability with assumptions. For example, whether the CNF $(t_{r1c1}) \wedge (\neg t_{r1c2}) \wedge (\neg t_{r1c3}) \wedge (t_{r1c1} \vee t_{r1c2}) \wedge (\neg t_{r1c3})$ is satisfiable when $t_{r1c1} = false$? As is easy to see, the CNF is satisfiable in general, but unsatisfiable with the given assumption. From this point we conclude, that t_{r1c1} cannot be false, and the cell $r_1 - c_1$ cannot contain N. Checking the satisfiability with assumption $t_{r1c1} = true$ returns the positive answer. Thus, we identified that the cell $r_1 - c_1$ contains T. Then we perform the questioning of the oracle for each variable, that gives us the full picture. The results of such questioning are presented in Table 3.9.

Table 3.9: Oracle questioning

Variable	SAT if T?	SAT if F?	Result
t_{r1c1}	+	-	T
t_{r1c2}	-	+	N
t_{r1c3}	-	+	N

We observe the same result as given by the analysis based on propagation rules, what is of course not surprisingly because we use the same semantic, and only representation of the input and method of searching the result are different.

Now we can formalize all manipulations as algorithms. We assume that we can unambiguously represent each cell of the matrix as a propositional variable and restore afterwards for each variable the cell of the matrix. For example, if we enumerate each row and each column of the matrix starting from 0, then it is possible to calculate *id* of the variable as $var\_index = row\_index * number\_of\_columns + column\_index$. The CNF contains the list of all possible variables. Number of variables in the list corresponds the number of cells in the matrix. Each variable has a flag determining whether it occurs in CNF or not. After analysis, we check

variables for each cell and keep the cell empty if variable did not take part in the reasoning, otherwise the content of cell corresponds the set of allowed variable values (T, F, T *and* F).

Algorithm 2 Converting artifact groups and no-trace relations into CNF

Input: Set of artifact groups and no-trace relations
Output : CNF

```
1:  cnf ←< EMPTY CNF >                          ▷ initialize with number of matrix cells
2:  for all AG(c, {r1, r2, ...}) do
3:      clause ←< EMPTY CLAUSE >
4:      for all rfrom{r1, r2, ...} do
5:          var ← getVariableIndex(r, c);
6:          clause.addVariable(var);
7:          cnf.setActive(var);
8:      end for
9:      cnf.addClause(clause);
10: end for

11: for all AG(r, {c1, c2, ...}) do
12:     clause ←< EMPTY CLAUSE >
13:     for all cfrom{c1, c2, ...} do
14:         var ← getVariableIndex(r, c);
15:         clause.addVariable(var);
16:         cnf.setActive(var);
17:     end for
18:     cnf.addClause(clause);
19: end for

20: for all no − trace(r, c) do
21:     clause ←< EMPTY CLAUSE >
22:     var ← getVariableIndex(r, c);
23:     clause.addVariable(−var);                ▷ negated variable for no-trace
24:     cnf.setActive(var);
25:     cnf.addClause(clause);
26: end for
```

For the trace analysis we need a list of oracle answers, one for each propositional variable from CNF. Such answers may contain one of three values : T (CNF is satisfiable, only if the corresponding variable is equal to true), F (CNF is satisfiable, only if the corresponding variable is equal to false) and TF (CNF is satisfiable in both cases). The number of answers is the same as the number of variables; thus we can use the same indices. Then the trace analysis may be described by algorithm 3

For the described approach suffice a SAT solver, that is only able to identify the satisfiability. In the next chapter, we show how to improve the algorithm, such that single oracle check provides all necessary data for the analysis. In this case, we also need an assignment of literals, therefore

Algorithm 3 Trace analysis

Input: CNF
Output : Filled trace matrix

```
1: answers ←< EMPTY LIST >                    ▷ initialize with number of variables
2: for all active variable v of CNF do
3:     if CNF.isSatisfiable(v = true) then     ▷ with the assumption v=true
4:         answers[v].setT();
5:     end if
6:     if CNF.isSatisfiable(v = false) then    ▷ with the assumption v=false
7:         answers[v].setF();
8:     end if
9: end for
   /*Fill up the matrix*/
10: for all row with rowIndex, column with columnIndex do
11:     varIndex = rowIndex ∗ numberOfColumns + columnIndex;
12:     if CNF.isVariableActive(varIndex) then
13:         if answers[varIndex] == T then
14:             cells[rowIndex, columnIndex] = T;
15:         else if answers[varIndex] == F then
16:             cells[rowIndex, columnIndex] = N;
17:         else
18:             cells[rowIndex, columnIndex] =< EMPTY >;
19:         end if
20:     end if
21: end for
```

a full solver. Examples of such solvers are PicoSAT[1], MiniSAT[2], SAT4J[3]. In the present work we use the PicoSAT solver from Armin Biere, because it provides not only an assignment of literals if CNF is satisfiable, but also information about conflict contributors if CNF is unsatisfiable using HUMUS approach that is described in the chapter ????.

3.3 Conflicts resolution strategies

The described technique of trace analysis only works under the assumption that the input is consistent and does not cause conflicts. An example of such conflict may be a situation when we derive trace and no-trace relation for the same pair of requirements and codes.

Let us consider the input for the matrix showed in Table 3.7:

`Dependency 1:` {`CrO`} `ImplExactly` {`OD`}

`Dependency 2:` {`CrO`} `ImplAtMost` {`OD, OL`}

[1] http://fmv.jku.at/picosat
[2] http://minisat.se/
[3] http://www.sat4j.org/

We showed in the previous chapter that this input is consistent (conflict free), and analysis can be performed. What will happen if we add a new input: `Dependency 3 : {CrO} ImplNot {OD}` ? From the third dependency based on the semantic of $ImplNot$, we produce one no-trace relation $no-trace(CrO, OD)$ that contradicts with $trace(CrO, OD)$ derived from the artifact group $AG(CrO, \{OD\})$ of the Dependency 1 and violates one of the correctness constraints. In term of CNF, we have a conflict, expressed in its general unsatisfiability. The CNF will take the form $(\neg t_{CrO,OD}) \wedge (t_{CrO,OD}) \wedge (\neg t_{CrO,OL}) \wedge (\neg t_{CrO,CA}) \wedge (t_{CrO,OD} \vee t_{CrO,OL}) \wedge (\neg t_{CrO,CA})$ in which the two first clauses result in unsatisfiable CNF. This means that SAT reasoning can not be performed, and we can not get any information about other variables.

In real life, we can not avoid conflicts and inconsistencies in input data especially for large systems. For instance, if more than one developer work simultaneously on the model, they may easily provide conflicting data. By the same way, we can not avoid conflict while maintaining traceability information. If a model or code of the system change, the traceability information has to be updated. However, the knowledge on some system parts may be outdated, or key personnel may have moved. This implies the ability to live with conflicts and to handle them properly.

Finding traceability information we can consider as a decision making process, in which we identify for each requirement and code whether they trace or not, so we deal with Boolean decisions. Working on a higher level of abstraction (dependencies) is the decision making process (but not Boolean) too, as the engineer adds dependencies incrementally and decides on each step which type of dependency to apply to artifacts. After each step, an engineer can see the impact of his/her decisions, and once a conflict occurs it must be resolved, otherwise the reasoning terminates.

A. Nöhrer and A. Egyed described in [19] conflict resolution strategies during decision making that can be considered as useful for the problem, described in the present work. First of all they distinguish two basic cases: 1) no valid configuration exists, 2) valid configuration exists. Obviously the correct and complete traceability information always exists theoretically if we consider it for a real system or a system under development, so we are interested in the second case. Here, a conflict occurs if the input is not consistent; for example if some subset of dependencies contains errors. We understand under the consistent input the input that does not cause the general unsatisfiability for constructed CNF. Obviously, the consistent input may not be correct if it does not reflect system precisely. In the present work we focus on the consistency of the input and do not verify its conformance with the real system (for instance, through dynamic analysis of the system during execution).

The first interesting strategy is *Fix right away.* The strategy may be applied at the exact moment the user introduce an inconsistency by adding a dependency conflicting with the previous input.

In this case the input will be immediately returned into a conflict free state and will never be inconsistent; thus the reasoning will not terminate. The fix right away strategy may be in one's turn divided into the following types:

- **Single Undo**. The simplest way to return the input into a consistent state is to retract the last dependency that caused the conflict and to ask a user to make another decision. Such strategy may be of course inappropriate if the user is sure about the correctness of the last dependency and wants to keep it. From the other hand if the user is not well familiar with the system this strategy may be an acceptable solution. Applying Single Undo to the described example we remove the *Dependency 3* from the input that results in a conflict free state.
- **Sequential Undo**. This strategy is useful in cases when the user assumes the last dependency to be correct, and this dependency must be preserved. In this case the problem should be in the previous dependencies. The idea is to retract dependencies until the most recent one does not cause a conflict any more. The obvious disadvantage of this approach is that "good" dependencies may be also retracted which is not desirable. What will be removed depends on the input order, so we can observe different result for the different orders of the same set of dependencies. In the example *Dependency 2* and *Dependency 1* must be retracted, although the *Dependency 1* suffice. To avoid this *Selective Undo* may be applied.
- **Selective Undo**. The idea is to retract exactly input part causing the conflict. To identify them, specific reasoning techniques mentioned in [19] may be applied. However in many scenarios it can not be performed automatically if, for instance, we identify a group of dependencies as a conflict source but retracting only one of them is sufficient. Either all of the dependencies from this group will be deleted (or random subgroup resulting in conflict free state) or the user is asked to select dependencies to be removed manually, where the risk of the incorrect answer can not be excluded.

What if it is not desired to resolve the conflict immediately, for instance, if a user wants to continue the work and resolve conflict later or it is not obvious how to fix the conflict? Additional information provided after conflict occurrence may, for example, help to decide which dependencies are erroneous. The idea is to provide the possibility to continue work even if the input is inconsistent. Approaches to the solution of the problem described A. Nöhrer and A. Egyed in [19, 20].

The simplest approach is *Continue manually*. Since the reasoning is complex in the presence of conflicts, it is possible to allow the user to perform input without such reasoning. For the traceability analysis it is not a vital solution as the main goal of the analysis can not be reached.

Another weakness of this method is impossibility to identify new conflict caused by new input if the reasoning is disabled.

There is a suitable approach called *Tolerating inconsistencies*[20]. It allows to handle inconsistencies during decision making process based on SAT-based reasoning and can be applied to the traceability problem. The idea is to exclude the part of the input that contributes a conflict from the reasoning, such that a SAT-solver would be able to continue work and keep this part isolated to further resolving. In case of SAT-based reasoning for the traceability problem there are no inconsistencies as long as the CNF can be evaluated to true. If a conflict occurs, the CNF will be always $UNSAT$, and addition of new dependencies will make no effect on the result, and the analysis is lost. The solution is to tolerate inconsistencies. Tolerate in this case does not mean to fix them. We do not change the input, we only isolate a part of the input (separate it from other parts, but do not delete it) to make CNF satisfiable and continue reasoning and let the user decide, when to fix the conflict. The next part describes existing strategies for conflict isolation:

- **Disregard All** is the simplest and trivial strategy that isolates all dependencies so far, that means that all input before the conflict is isolated from the future reasoning. The strategy guarantees that the erroneous part will be isolated; however, the meaningfulness of isolation of all dependencies is doubtful.
- **Skip Strategy** isolates the last added dependency. As isolation occurs only in case of conflicts, the dependency contributing the conflict will be isolated immediately, and the input so far remains consistent. This strategy may force the user to fix the conflict immediately if he or she is sure about the correctness of the isolated dependency and needs to have the reasoning results including this dependency.
- **MaxSAT Strategy** stands for the *maximal satisfiability* and applied to the problems expressed by CNF. The basic concept is to identify the subset of clauses of the CNF with maximal cardinality, such that conjunction of clauses from this subset results in the satisfiable CNF. In the context of the trace analysis, we have to select as many dependencies as possible, such they produce consistent input. All dependencies not contained in this set are isolated. The weakness of the MaxSAT it that it simply identifies the minimal subset of dependencies to be removed from the input; therefore, the result is not deterministic and more than one final subsets with the same cardinality may be produced. For the considered example *Dependency 1* or *Dependency 3* can be isolated. Additionally there is absolutely no guarantee that the erroneous dependency will be disabled; thus it may cause conflicts with future input that could be avoided if this dependency would be isolated.
- **HUMUS** stands for *High Level Union of Minimal Unsatisfiable Sets*. This concept is applicable only on high level; thus we have to work with assumptions. The implementation

of the HUMUS strategy takes a shortcut in comparison to the approach of Liffiton[21] to compute all *MUSes* (*Minimal unsatisfiable set* or *MUS* is the subset of assumptions that evaluate the CNF to $UNSAT$, but removing any single assumption from this subset makes CNF satisfiable) and their union. By this way the method finds all direct and indirect contributors of conflict, what guarantees the identification of erroneous part. For the illustrative example the *Dependency 1* and *Dependency 3* will be selected as both of them are related to the conflict.

One can see, that all strategies produce different sets of dependencies to be isolated; thus, we need to find the most appropriate one that will satisfy our goals. We can assess them by the following criteria:

- **Incompleteness** : An isolation strategy removes with high probability correct dependencies from the input. That leads to incomplete reasoning meaning that we lose some information. Generally, it is not trivial to assess the completeness of input without validation as we do not know how should look like the resulting trace matrix. However we need to tend to minimize potential information loss. *Disregard All* strategy shows the most incomplete reasoning, whereas *MaxSAT* and *Skip* strategy isolate the minimal information. For the *HUMUS* the degree of incompleteness may vary between best and worst cases depending on the input.
- **Incorrectness** : Incorrect reasoning is the result of reasoning with defects. Incorrect reasoning may cause the isolation of correct dependencies and skipping erroneous ones. If a conflict occurs, a strategy tries to bring the CNF in SAT state and may keep dependencies active despite the facts that they are incorrect. This potentially leads to new conflicts with future input that would not appear if erroneous dependencies have been isolated. When using *HUMUS* or *Disregard All* strategy one can be sure that an error will be eliminates, as *Disregard All* simply isolates all dependencies so far, and *HUMUS* identifies all dependencies direct and indirect related to the conflict. In the worst case *HUMUS* may isolate all dependencies like *Disregard All* strategy if all of them are contributors of error. On the other hand *MaxSAT* and *Skip* strategies do not necessarily remove dependencies the user will remove, and one can not be sure any more that all erroneous dependencies are isolated.

The trace analysis requires the assurance that the user continues work without errors; therefore, we can not use *MaxSAT* and *Skip* strategies, though they provide minimal incompleteness by isolating minimal subset of dependencies. *Disregard All* strategy is the worst case in term of incompleteness and so we can conclude that it is most reasonable to use the *HUMUS* strategy.

The implementation of the HUMUS strategy takes a shortcut in comparison to the approach of Liffiton[21] to compute all $MUSes$. In the present work we use the *HUMUS* implementation described in [22] that calculates *MSSes* using assumptions over clause selector variables. *MSS* stands for *maximal satisfiable set* that is a set of assumptions, such that CNF is satisfiable, and the set is of the maximal size it can be. If such subset is found its complement is *MCS* (*minimal correcting set* : the minimal subset of assumptions, such that removing it from the reasoning always result in the satisfiable CNF). *HUMUS* in the implementation we use identifies all $MSSes$ for the given set of assumptions and, therefore, all $MCSes$. The union of all $MCSes$ is the same subset as the union of all $MUSes$, and we can consider it as the set of assumption contributing an error. For the trace analysis problem *HUMUS* provides a set of dependencies contributing an error. The *HUMUS* strategy allows to isolate conflicts on high level assuming that the CNF itself is satisfiable, but the current set of assumptions makes it unsatisfiable. For example, the CNF $(a \vee b \vee c) \wedge (\neg a \vee \neg c) \wedge (\neg b)$ is satisfiable in general but $UNSAT$ with assumptions $a = true$, $c = true$. Isolated are assumptions that cause this unsatisfiability (for example, we disable the assumption $c = true$, and CNF becomes SAT). The used model does not support assumption directly because the CNF is not static but is changed every time as new input is added, some dependency is modified or removed.

To support assumptions we introduce clause selector variables. Since clauses in CNF are disjunctions, adding a variable to be used as a selector is easy, if the clause should be ignored true is assumed for the variable which results in the clause being true and not being influence the result.

Consider the following inputfor the matrix showed in Table 3.7:

```
Dependency 1:  {CrO} ImplExactly {OD}
Dependency 2:  {CrO} ImplAtMost {OD, OL}
Dependency 3:  {CrO} ImplNot {OD}
```

For each dependency we derive set of artifact groups and no-trace relation that will be converted in clauses:

{*CrO*} *ImplExactly* {*OD*}

AG(CrO, {OD})

AG(OD, {CrO}) $\rightarrow$ $(t_{CrO,OD})$

no-trace(CrO, OL) $\rightarrow$ $(\neg t_{CrO,OL})$

no-trace(CrO, CA) $\rightarrow$ $(\neg t_{CrO,CA})$

{*CrO*} *ImplAtMost* {*OD, OL*}

AG(CrO, {OD, OL}) $\rightarrow$ $(t_{CrO,OD} \vee t_{CrO,OL})$

no-trace(CrO, CA) $\rightarrow$ $(\neg t_{CrO,CA})$

$\{CrO\}$ *ImplNot* $\{OD\}$

no-trace$(CrO, OD) \quad \rightarrow \quad (\neg t_{CrO,OD})$

We associate each dependency of the input with a selector variable and add this variable to the clauses derived from it. For the three dependencies about we introduce three variables s_1, s_2 and s_3 accordingly for dependencies 1, 2 and 3. Result is presented in the Table 3.10.

Table 3.10: Clause selector variables insertion

Before insertion	**After insertion**
$(t_{CrO,OD})$	$(t_{CrO,OD} \vee s_1)$
$(\neg t_{CrO,OL})$	$(\neg t_{CrO,OL} \vee s_1)$
$(\neg t_{CrO,CA})$	$(\neg t_{CrO,CA} \vee s_1)$
$(t_{CrO,OD} \vee t_{CrO,OL})$	$(t_{CrO,OD} \vee t_{CrO,OL} \vee s_2)$
$(\neg t_{CrO,CA})$	$(\neg t_{CrO,CA} \vee s_2)$
$(\neg t_{CrO,OD})$	$(\neg t_{CrO,OD} \vee s_3)$

It is easy to see that if a clause selector variable is $false$, all clauses containing this variable will be equivalent to their original state according the law $x \vee F = x$. In this case we say that corresponding dependency is active. If a selector variable is $true$, then all clauses containing this variable can be ignored as they will be always evaluated to true according $x \vee T = T$. This mechanism allows to enable or disable (isolate) dependencies by simple assignment $true$ or $false$ to corresponding selector variables. As the set of assumption for *HUMUS* we take the set $\{s_1 = false, s_2 = false, s_3 = false\}$ expressing that every dependency is initially active. If the SAT-solver detects *UNSAT*, we apply *HUMUS* to the set of assumptions. *HUMUS* returns a subset contributing a conflict. For the given example input the result will be $\{s_1 = false, s_3 = false\}$, so we need to isolate dependencies 1 and 3, what can be done by assignment true to s_1 and s_3. After this operation CNF takes a form: $(t_{CrO,OD} \vee T) \wedge (\neg t_{CrO,OL} \vee T) \wedge (\neg t_{CrO,CA} \vee T) \wedge (t_{CrO,OD} \vee t_{CrO,OL} \vee F) \wedge (\neg t_{CrO,CA} \vee F) \wedge t_{CrO,OD} \vee s_3)$ that is equivalent to $(t_{CrO,OD} \vee t_{CrO,OL} \vee F) \wedge (\neg t_{CrO,CA} \vee F)$.

Table 3.11: Mobile Waiter trace matrix after analysis

	01: CrO	02: ClO
OrdersData(OD)		
Offline(OL)		
Calculator(CA)	N	

Resulting trace matrix after the isolation is presented in Table 3.11. Comparing to the matrix in Table 3.8 we have lost data for cells $OD - CrO$ and $OL - CrO$. The cause of the data loss is the unavoidable incomplete reasoning caused by *HUMUS*. However, we still avoid incorrect reasoning. *Disregard All* strategy would result in this case in the state, where every dependency is isolated, that is the worst case. More detailed analysis of correctness for HUMUS we do in the chapter 5.

Dependencies 1 and 3 is now isolated but not removed, so we can use them for conflict resolution. The dependencies not related to the conflict remain active, and CNF is SAT, so the SAT-solver is able to continue the work and we do not lose the automation.

Obviously not the whole isolated dependency may cause an inconsistency, but rather only some artifact groups may be involved in the conflict. So one can perform isolation of only a part of the dependency that results in smaller information lost while isolation. This can be useful in situations, where the engineer provides partially correct dependencies. However, even in this case we isolate groups of cells in the matrix, where as the isolation of individual cells may be sufficient. Moreover, isolation of individual cells together with involved dependencies may provide a strong basis for conflict resolution. So it is also reasonable to perform isolation on the cell level. The exact algorithms for different isolation strategies will be describe in the next chapter.

Chapter 4

Approach

In this chapter we explain detailed all steps of our implementation of trace analysis algorithm using SAT-based reasoning, isolation strategies on units level and cells level, incremental reasoning, user guidance based on the incremental reasoning, and demonstrate the improvement of the oracle algorithm described in the chapter 3.

We assume the input of the approach as a set of dependencies. Generally the order of the dependencies may play a role and may be used for the user guidance about erroneous dependencies that is described in the third subsection of this chapter. For the time moment we assume that the order is not relevant. Then the steps of the trace analysis are the following:

1. Deriving of the of artifact groups and no-trace relations from the set of input dependencies.
2. Selection of isolation strategy.
3. Transformation the input into CNF.
4. Reasoning using SAT-solver.
5. Filling up the traceability matrix.

The algorithm 1 in the previous chapter describes the process of deriving artifact groups (AG) and no-trace relations(NTR) (we use the term *units*) for single dependency. By the repeating of the algorithm for all dependencies we can obtain the necessary data. Assume we have a set of all dependencies $Dependencies$ that is the input for the first step. We need to fill up the sets $ArtifactGroups$ and $NoTraceRelations$ that are output of the first step. We need to have an access to the list of all code elements (C) and requirements (R), as it is necessary for the processing of $ImplExactly-$ and $ImplAtMost-$ dependencies. The following algorithm describes the process:

Algorithm 4 Deriving artifact groups and no-trace relations

1: **for all dependency** $from$ **Dependencies do**
2: $\{AG_{dependency}, NTR_{dependency}\} \leftarrow derive(dependency, C, R)$; ▷ by algorithm 1
3: $ArtifactGroups \leftarrow ArtifactGroups \cup AG_{dependency}$;
4: $NoTraceRelations \leftarrow NoTraceRelations \cup NTR_{dependency}$;
5: **end for**

4.1 Isolation strategies

Before we begin the transformation of derived elements into clauses, we need to decide which isolation level to use. The lower the granularity of the isolation, the less information we lose. We are able also to find conflict contributors more accurate. We described above, how to isolate dependencies by the addition of selector variables to the clauses. To isolate individual artifact groups or no-trace relations we can introduce selector variables not unique to each dependency, but rather unique for each unit to be potentially isolated. Table 3.10 describing the result of the addition of selector variables for the given example input will be then modified to the Table 4.1

Table 4.1: Clause selector variables on units level

Before insertion	**After insertion**
$(t_{CrO,OD})$	$(t_{CrO,OD} \vee s_1)$
$(\neg t_{CrO,OL})$	$(\neg t_{CrO,OL} \vee s_2)$
$(\neg t_{CrO,CA})$	$(\neg t_{CrO,CA} \vee s_3)$
$(t_{CrO,OD} \vee t_{CrO,OL})$	$(t_{CrO,OD} \vee t_{CrO,OL} \vee s_4)$
$(\neg t_{CrO,CA})$	$(\neg t_{CrO,CA} \vee s_5)$
$(\neg t_{CrO,OD})$	$(\neg t_{CrO,OD} \vee s_6)$

As the set of assumption for *HUMUS* we consider the set $\{s_i = false,\ i = 1 \dots 6\}$ expressing that every unit is initially active. If the SAT-solver detects $UNSAT$, we apply *HUMUS* to the set of assumption. *HUMUS* returns a subset contributing a conflict. In case of unit level isolation the set $\{s_1 = false,\ s_2 = false,\ s_4 = false,\ s_6 = false\}$ will be obtained, so we need to isolate artifact groups related to the clauses 1, 2, 4, 6 what can be done by the assignment *true* to s_1, s_2, s_4, s_6. After this operation CNF takes a form: $(t_{CrO,OD} \vee T) \wedge (\neg t_{CrO,OL} \vee T) \wedge (\neg t_{CrO,CA} \vee F) \wedge (t_{CrO,OD} \vee t_{CrO,OL} \vee T) \wedge (\neg t_{CrO,CA} \vee F) \wedge (t_{CrO,OD} \vee T)$ that is equivalent to $(\neg t_{CrO,CA} \vee F) \wedge (\neg t_{CrO,CA} \vee F)$. Resulting trace matrix after the isolation is presented in Table 4.2.

Table 4.2: Mobile Waiter trace matrix after analysis

	01: CrO	02: ClO
OrdersData(OD)		
Offline(OL)		
Calculator(CA)	N	

As we identified the selector variables related to the conflict, we are able to identify units to be isolated. In this case we need to keep the order of clauses of the CNF. The selector variables (and therefore assumptions) can be enumerated, and we will be able to identify affected clauses. Units may be also captured in ordered lists, so we do not need to preserve the mapping clause=¿unit. Actually we do not even need to determine clauses to be isolated if the order of units corresponds the order of clauses. For the given example, having the list of units

1. AG(OD, $\{CrO\}$) $\rightarrow$ $(t_{CrO,OD})$
2. no-trace(CrO, OL) $\rightarrow$ $(\neg t_{CrO,OL})$
3. no-trace(CrO, CA) $\rightarrow$ $(\neg t_{CrO,CA})$
4. AG(CrO, $\{OD, OL\}$) $\rightarrow$ $(t_{CrO,OD} \vee t_{CrO,OL})$
5. no-trace(CrO, CA) $\rightarrow$ $(\neg t_{CrO,CA})$
6. no-trace(CrO, OD) $\rightarrow$ $(\neg t_{CrO,OD})$

and the HUMUS output $\{s_1 = false, s_2 = false, s_4 = false, s_6 = false\}$ one identifies the list of affected units:

1. AG(OD, $\{CrO\}$) $\rightarrow$ $(t_{CrO,OD})$
2. no-trace(CrO, OL) $\rightarrow$ $(\neg t_{CrO,OL})$
3.
4. AG(CrO, $\{OD, OL\}$) $\rightarrow$ $(t_{CrO,OD} \vee t_{CrO,OL})$
5.
6. no-trace(CrO, OD) $\rightarrow$ $(\neg t_{CrO,OD})$

What we have to preserve is the mapping $Unit \rightarrow Dependency$ or $UnitID \rightarrow Dependency$ to identify dependencies contributing a conflict. If only a part of the dependency is isolated, it can be easier to understand how the dependency must be modified to become conflict free. For the current example all three dependencies are affected.

The next level of granularity is the **cell level**. Here, we try to identify which cells are affected by a conflict. In terms of dependencies that means that if one can conclude from some dependency that a cell contains T (not empty, not N, not TN), there exists another dependency that provides N for the same cell. In terms of CNF that means that one clause forces the variable related to the cell to be equal to $true$ while some other clause forces this variable to be $false$. How we identify such variables?

Analyzing the conversion procedure one can understand that a conflict is possible if some clause forces some variable to be $false$. In other cases if there are no such clauses, conflict can not occur. Artifact groups always produce clauses with the form $(v_1 \vee v_2 \vee \ldots \vee v_k)$ in which each literal is without negation. No-trace relations produce clauses of the form $(\neg v)$. As we convert only artifact groups and no-trace relations into clauses, the CNF consist of clauses of only these two types. That implies we can observe forced $false$ given only by a clause of the form $(\neg v)$. On

the other hand if a conflict occurs, it always covers such clauses, because conflicts are impossible for CNF built for the traceability problem without forced $false$. Moreover, each clause derived from no-trace relations always contains only single literal and, therefore, represents a single cell. If we could identify all no-trace clauses involved into the conflict, we get in this case all cells involved into the conflict and can perform isolation on the cell level. One can reach this by adding selector variables only to the clauses for no-trace relations. *HUMUS* will identify such clauses and return the list of variables that represent, in fact, conflicting cells.

To bring the CNF to the SAT state again, it is also sufficient for the case of isolation on *cells level* to isolate the involved clauses derived for no-trace relations. Discovered cells we mark as *"isolated"*. It is clear that according this isolation scheme other artifact groups delivering information for isolated cells are not isolated. However for each such artifact group we ignore the values of variables related to the isolated cells. This ignoring does not take an influence on the result. If the cell is isolated it occurs in at least one artifact group and one no-trace relation. From the artifact group the corresponding variable may only take the true value; otherwise CNF is UNSAT. Other variables of the clause related to this AG are not allowed to vary; otherwise we could assign to the *"isolated"* variable $false$ value, and, by varying of values of some other variables of this clause, bring CNF into the SAT state. If we isolate the no-trace relation/relations containing this variable and make CNF satisfiable we allow the *"isolated"* variable to be $true$ without influencing values of other variables and, therefore, without bothering the final result. The cell of the trace matrix, represented by this variable will not contain T (as follows from the reasoning), but rather will be marked as *"isolated"* or can also contain C (*conflict*).

The isolation itself gives us the list of no-trace relation, related to the conflict. They may be part of $ImplAtMost-$, $ImplExactly-$ or $ImplNot-$ relations. However, we can not identify the affected artifact groups and, therefore, the dependencies containing these artifact groups, that is of course valuable because they are also a part of the conflict. The problem may be solved by keeping the index of the artifact groups or/and dependencies related to each cell potentially may be involved into conflict (that means for all cells contained in some no-trace relation) or by direct search of the variable occurrence in the set of dependencies or artifact groups.

Of course if we obtain no-trace clauses with the same literal from multiple different dependencies, we add to them the same selector variable. Result is presented in the Table 4.3.

As the set of assumption for *HUMUS*, we consider the set $\{s_1 = false,\ s_2 = false,\ s_3 = false\}$ expressing that every unit is initially active. If the SAT-solver detects $UNSAT$, we apply $HUMUS$ to the set of assumptions. $HUMUS$ returns a subset contributing a conflict. In case of unit level isolation the set $\{s_3 = false\}$ will be obtained, so we need to isolate no-trace relation related to the clauses with the selector variable s_3 what can be done by assignment $true$ to s_3. After this operation, CNF takes a form: $(t_{CrO,OD}) \wedge (\neg t_{CrO,OL} \vee F) \wedge (\neg t_{CrO,CA} \vee$

TABLE 4.3: Clause selector variables insertion

Before insertion	**After insertion**
$(t_{CrO,OD})$	$(t_{CrO,OD})$
$(\neg t_{CrO,OL})$	$(\neg t_{CrO,OL} \vee s_1)$
$(\neg t_{CrO,CA})$	$(\neg t_{CrO,CA} \vee s_2)$
$(t_{CrO,OD} \vee t_{CrO,OL})$	$(t_{CrO,OD} \vee t_{CrO,OL})$
$(\neg t_{CrO,CA})$	$(\neg t_{CrO,CA} \vee s_2)$
$(\neg t_{CrO,OD})$	$(\neg t_{CrO,OD} \vee s_3)$

$F) \wedge (t_{CrO,OD} \vee t_{CrO,OL}) \wedge (\neg t_{CrO,CA} \vee F) \wedge t_{CrO,OD} \vee T)$ that is equivalent to $(t_{CrO,OD}) \wedge (\neg t_{CrO,OL}) \wedge (\neg t_{CrO,CA}) \wedge (t_{CrO,OD} \vee t_{CrO,OL}) \wedge (\neg t_{CrO,CA})$

This CNF implies that the cell $CrO - OD$ must contain T, but this cell was isolated; therefore, we ignore all values given by CNF for this cell. Result is presented in Table 4.4 (C stands for the conflict).

TABLE 4.4: Mobile Waiter trace matrix after analysis

	01: CrO	02: ClO
OrdersData(OD)	C	
Offline(OL)	N	
Calculator(CA)	N	

So one can see, that we have lost the minimum information and obtained conflicting cell. Analysis of all dependencies related to this cell may help to identify an error.

The selected level of isolation will determine how we add the selector variables to clauses. In all cases, we need to have the mapping between selector variables and dependencies or units they represent. Even if the order of elements is irrelevant for the reasoning, we assume it fixed to the reasoning time. This assumption provides the ability to identify elements related to the conflict unambiguously.

To add a selector variable to the clause we use the method `getSelectorVariable (isolationStrategy, unit)`, that depends on `isolation strategy` (or isolation level) and the `unit` to be transformed into the clause. The method requires for the correct reasoning an ordered list of dependencies *Dependencies* and an ordered list of units *Unit*s containing both artifact groups and no-trace relation. We also need for the isolation on the cells level the list of no-trace relations NTR. The method `getSequenceNumber (object, list)` allows to identify the number of elements in the list.

The index of selector variable in CNF may not fully match the logical index of selector variable. For the correct mapping we use `getVariableByIndex` method, that depends on the concrete implementation of CNF. In case of isolation on cells level, we also use $ZERO\_SELECTOR$. This is the index of the selector variable added to all artifact group clauses. At the end, we add

Algorithm 5 getSelectorVariable method

```
1: function getSelectorVariables(isolationStrategy, unit)
2:     if isolationStrategy == ISOLATE_DEPENDENCIES then
3:         dependency ← getParentDependency(unit);
4:         selectorVariableIndex ← getSequenceNumber(dependency, Dependencies);
5:     else if isolationStrategy == ISOLATE_UNITS then
6:         selectorVariableIndex ← getSequenceNumber(unit, Units);
7:     else                                                    ▷ ISOLATE_CELLS
8:         if unit is artifact group then
9:             selectorVariableIndex ← ZERO_SELECTOR
10:        else
11:            selectorVariableIndex ← getSequenceNumber(unit, NTR);
12:        end if
13:    end if
14:    return getVariableByIndex(selctorVariableIndex);
15: end function
```

a clause to the CNF making artifact groups clauses always active ($\neg ZERO\_SELECTOR$). We need this selector neither for the reasoning nor the isolation but use it to keep the structure of the clause consistent, assuming that every clause always has exactly one selector variable that is the last variable of the clause.

Then the algorithm 2 of transformation input to CNF take a form of the algorithm 6.

In this case the CNF contains not only the list of all possible variables representing cells of the matrix but also the list of all selector variables.

4.2 Oracle improvement

Having constructed CNF, we are ready to perform the reasoning. We described the common principle of the reasoning in the previous chapter. This method requires N satisfiability checks are being performed by the oracle if we have N different variables (excluding selector variables). However if we carefully analyze the structure of the constructed CNF we can modify the checking step to only single check.

The oracle builds one answer for a variable v according the following principle:

1. Check satisfiability for the assumption $v = T$
2. Check satisfiability for the assumption $v = F$
3. If SAT($v = T$) then $cell = T$
4. If SAT($v = F$) then $cell = N$

Algorithm 6 Converting artifact groups and no-trace relations into CNF

Input: Set of artifact groups and no-trace relations, isolation strategy
Output : CNF

```
1: cnf ←< EMPTY CNF >                          ▷ initialize with number of matrix cells
2: for all AG(c, {r1, r2, ...}) as ag do
3:     clause ←< EMPTY CLAUSE >
4:     for all rfrom{r1, r2, ...} do
5:         var ← getVariableIndex(r, c);
6:         clause.addVariable(var);
7:         cnf.setActive(var);
8:     end for
9:     selector ← getSelectorVariable(isolationStrategy, ag);
10:    clause.addVariable(selector);
11:    cnf.setActive(selector);
12:    cnf.addClause(clause);
13: end for

14: for all AG(r, {c1, c2, ...}) as ag do
15:    clause ←< EMPTY CLAUSE >
16:    for all cfrom{c1, c2, ...} do
17:        var ← getVariableIndex(r, c);
18:        clause.addVariable(var);
19:        cnf.setActive(var);
20:    end for
21:    selector ← getSelectorVariable(isolationStrategy, ag);
22:    clause.addVariable(selector);
23:    cnf.setActive(selector);
24:    cnf.addClause(clause);
25: end for

26: for all no − trace(r, c) as ntr do
27:    clause ←< EMPTY CLAUSE >
28:    var ← getVariableIndex(r, c);
29:    clause.addVariable(−var);                ▷ negated variable for no-trace
30:    cnf.setActive(var);
31:    selector ← getSelectorVariable(isolationStrategy, ntr);
32:    clause.addVariable(selector);
33:    cnf.setActive(selector);
34:    cnf.addClause(clause);
35: end for
```

5. if SAT($v = F$) and SAT($v = T$) then $cell = EMPTY$

We don't consider the case where CNF is $UNSAT$ for $v = F$ and $v = T$ because in this case the CNF is $UNSAT$ in general and we apply the isolation strategy.

Let us consider the case, where the CNF is SAT and draw attention to the following facts:

1. If a clause contains single variable (or two variables including selector), then the value of these variables in case of satisfiability of CNF may be only *true* if the variables is not negated and only $false$ if the variable is negated. This fact holds always, otherwise CNF would be unsatisfiable. That means that we do not need to check this variable by the oracle. We recognize such variables during clauses construction and activate so called *single bit* for them. For this purpose, we add a list of bits to the CNF; index of each bit corresponds the index of CNF variable (excluding selectors). If the variable is *single* in some clause, the corresponding bit is set to 1; otherwise it is 0. After the first checking of the CNF on satisfiability, the $PicoSAT$ solver returns a random set of variable assignments if the CNF is SAT. Although the set is random, this assignment make CNF satisfiable and variables with active single bit gets its correct final value (for example, if it is *true*, it will be always *true*), and we do not need to check them using oracle.

2. If some variable is allowed to be equal only to F, this variable occurs in at least one no-trace clause and has active single bit. This fact is in principle clear as artifact groups produce clauses of the form $(a \vee b \vee c \ldots)$. If such clause contains only one variable it may have only *true* value; but if the clause has more than one variable neither can be only$false$. Therefore, no-trace clauses are only contributors of $false$ (Ns in the matrix).

3. If some variable a is allowed to be equal only to *true*, it occurs either in a clause with one variable or in the clause like $(a \vee b \vee c \ldots)$, where for all variables except a no-trace clauses $(\neg b)$, $(\neg c)$,... exist. The fact is obvious for the first case. In the second case, if we assume that some of other clause variables is allowed to be not only $false$ but also *true*, the variable a is allowed to be $false$, because the clause may be evaluated to *true* even if $a = false$, what is a contradiction. That means that every variable $b, c, \ldots$ is forced to be $false$, and it is only possible if no-trace clauses for these variable exist. We organize another array of bits for such variables and call them *pseudo-single bits*.

4. All variables having inactive *single* or $pseudo - single$ bits may be equal *true* or $false$ and produce TN value for the corresponding matrix cell. This is correct according the conversion process of the CNF. The only contributors of $false$ are no-trace clauses, and variables of no-trace clauses (excluding selectors) always have active single bit (see fact 1). All other clauses never contain negated variables; therefore, they contribute *true* values if some clause contain single variables or if all variable of the clause except one are forced to be false (see fact 3). Thus, if the variable has an active *single* or active $pseudo - single$ bit, it will always have the value returned by the SAT-solver at first check, otherwise the variable will be T/F.

Using these facts it is enough to perform only first check to identify all possible values of variables. To prepare single bit we modify the algorithm 6, and add after each `cnf.addClause(clause);` statement the following:

```
1: if numberOfVars(clause)==1 then
2:     variable ← clause.getVarable(0);        ▷ index of variable on the position 0 in clause
3:     if variable is negated then                                          ▷ no-trace clause
4:         setNegativeSingleBit(variable);
5:     else
6:         setPositiveSingleBit(variable);
7:     end if
8: end if
```

We do not keep the common set of single bits for no-trace clauses and clauses of artifact groups, but save them in separate sets, as we need negative single bits for identifying pseudo-single bits. Identification takes place between the transformation step and the reasoning (see algorithm 7).

Algorithm 7 Identification of pseudo-single bits

```
1: for all clause in CNF do
2:     singleVariable ← 0;
3:     numberOfSingleVariables ← 0;
4:     for all variable in clause do
5:         if isSingleNegativeBitSet(variable) then
6:             numberOfSingleVariables + +;
7:         else
8:             singleVariable ← variable;
9:         end if
10:    end for
11:    if numberOfSingleVariables == clause.getNumberOfVariables - 2 then▷ all but
   one and selector
12:        setPseudoSingleBit(singleVariable);
13:    end if
14: end for
```

Now we are ready to perform the reasoning. We do not need to check every variable twice. Using single bits the reasoning may be organized as presented by algorithm 8. Recap, that now we consider the case, where CNF is satisfiable.

`fillMatrix()` method in the algorithm 8 represents a part of the algorithm 2 where the trace matrix is being filled according the oracle answers. This section remain unchanged. `isolate()` method implements the isolation strategy and described below. `CNF.isSatisfiable()` in the algorithm 8 checks the general satisfiability of the CNF with assumptions that all selector variables are equal to $false$. If the CNF is $UNSAT$, we apply $HUMUS$ to the set of assumptions (selector variables). It is irrelevant for the $HUMUS$ which isolation strategy is used because the CNF and assumptions always have the same structure. The only difference is the set of selector variables is analyzed by $HUMUS$ and how we interpret results. To apply $HUMUS$, we have to add an assumption for each selector variable. In principle `CNF.isSatisfiable()` in

Algorithm 8 Trace analysis with single bits

Input: CNF
Output : Filled trace matrix

```
1: answers ←< EMPTY LIST >                          ▷ initialize with number of variables
2: randomAssignment ←< EMPTY LIST >
3: if CNF.isSatisfiable() then                                              ▷ in general
4:     randomAssignment ← solver.getAssignment();
5: else
6:     isolate();
7:     randomAssignment ← solver.getAssignment();
8: end if
9: for all active variable v from CNF do
10:    if isSingleBitSetFor(v) then
11:        answers[v].set(v)                          ▷ set answer according value of v
12:    else
13:        answers[v].set("TN")
14:    end if
15: end for
16: fillMatrix();
```

the algorithm 8 may be presented as following:

```
1: function isSatisfiable
2:     assumptions ←< EMPTY LIST >;
3:     for all selector do
4:         assumptions.add(selector = false);
5:     end for
6:     return CNF.isSatisfiable(assumptions);
7: end function
```

As long as `CNF.isSatisfiable(assumptions)` returns $false$ one can immediately apply $HUMUS$ to get a set of assumptions contributing an error. By the assignment of $true$ value to the selector variables from the returned set we isolate incorrect clauses. By identifying elements (dependencies, units, cells) related to the isolated clauses, we provide an developer a set of possibly erroneous elements. The method `isolate()` from the algorithm 8 can be described as following:

```
1: function isolate
2:     conflictAssumptions ← solver.humus();
3:     for all assumption ∈ conflictAssumptions do
4:         selectorVar ← getSelectorVar(assumption);
5:         assumptions.remove(selectorVar = false);
6:         assumptions.add(selectorVar = true);
```

7: **end for**
8: $notifyUser(conflictAssumptions, isolaionStrategy)$;
9: **end function**

The `notifyUser()` method provides the user the set of elements isolated according the *isolationStrategy*. In other words the method returns the set of cells, related to each selector variable if the isolation is performed on the cell level, otherwise the set of artifact groups or dependencies. After the isolation the solver will return a random set of assumptions by the same way as in the case of conflict-free input because performed isolation guarantees the satisfiability of CNF.

4.3 Incremental reasoning

So far we described the algorithms for the trace analysis based on the provided dependencies. The order of the dependencies was relevant only for mapping of units to the clauses of CNF and identification of parent dependencies for those units. In other words the result of the trace analysis was independent of the dependencies order.

In normal scenario, an engineer may provide dependencies incrementally one by one. Assuming this scenario we can perform the reasoning in two ways:

- Provide some set of dependencies first. Activate the reasoning by request.
- Perform the reasoning and isolation after each added dependency.

The first scenario matches the approach described so far, in which we encode all provided dependencies into CNF for once (batch reasoning). The second approach more closely resembles the decision process, in which a user provides answers one by one. For example the online laptop configuration is the similar process. On each step the user has to answer a question by selecting one of the predefined possible answers. Very often the user must answer the questions in some predefined order, and only those choices of the remaining questions are presented, that are still available based on the already answered questions. This is normal because many answers are interrelated and not any combinations of them is relevant. If to allow the user to answer questions without control, the risk to get an invalid configuration becomes unapproprate. In this case the user is not allowed to select desired choice if it has been eliminated before, and the only option is to undo previous decisions. However, there is no guarantee that new answers will not lead to the same result. There are a lot of variants how to solve this conflict, and it is hard to identify questions to be revisited without tool support.

An alternative is to identify and revisit only decisions that are in conflict with the desired choice. This idea may be applied to conflicts in traceability configuration if an engineer provides

dependencies step by step. In principle it is the same decision process, only without predefined answers.

Let us consider and compare results for two processes (batch reasoning and incremental reasoning). Assume an engineer provides the following input for the trace matrix presented in Table 3.7:

`Dependency 1:` {`CrO`} `ImplExactly` {`OD`}

`Dependency 2:` {`CrO`} `ImplNot` {`OD, OL`}

`Dependency 3:` {`CrO`} `ImplAtMost` {`OD, OL`}

How do we perform the batch reasoning for the provided input? The first step is to derive units for the dependencies and transform them into clauses.

$\{CrO\}$ *ImplExactly* $\{OD\}$

$\text{AG}(CrO, \{OD\})$

$\text{AG}(OD, \{CrO\}) \quad \rightarrow \quad (t_{CrO,OD} \vee s_1)$

$\text{no-trace}(CrO, OL) \quad \rightarrow \quad (\neg t_{CrO,OL} \vee s_1)$

$\text{no-trace}(CrO, CA) \quad \rightarrow \quad (\neg t_{CrO,CA} \vee s_1)$

$\{CrO\}$ *ImplNot* $\{OD,\ OL\}$

$\text{no-trace}(CrO, OD) \quad \rightarrow \quad (\neg t_{CrO,OD} \vee s_2)$

$\text{no-trace}(CrO, OL) \quad \rightarrow \quad (\neg t_{CrO,OL} \vee s_2)$

$\{CrO\}$ *ImplAtMost* $\{OD,\ OL\}$

$\text{AG}(CrO, \{OD, OL\}) \quad \rightarrow \quad (t_{CrO,OD} \vee t_{CrO,OL} \vee s_3)$

$\text{no-trace}(CrO, CA) \quad \rightarrow \quad (\neg t_{CrO,CA} \vee s_3)$

The input is inconsistent, and the resulting CNF is unsatisfiable. Applying the *HUMUS* strategy for the set of assumptions $\{s_1 = false,\ s_2 = false,\ s_3 = false\}$ (all dependencies are active) we get the result $\{s_1 = false,\ s_2 = false,\ s_3 = false\}$ meaning that all provided dependencies are related to the conflict. The picture change if we add dependencies one by one and perform the reasoning after each step.

Step 1: Add *Dependency 1.* CNF takes a form $(t_{CrO,OD} \vee s_1) \wedge (\neg t_{CrO,OL} \vee s_1) \wedge (\neg t_{CrO,CA} \vee s_1)$. The CNF is satisfiable with the assumption $s_1 = false$, and the oracle return the answers : $\{t_{CrO,OD} = true, t_{CrO,OL} = false, t_{CrO,CA} = false\}$. The matrix takes in this case the form as it is showed in the table Table 4.5

Table 4.5: Mobile Waiter trace matrix after step 1

	01: CrO	02: ClO
OrdersData(OD)	T	
Offline(OL)	N	
Calculator(CA)	N	

Step 2: Add *Dependency 2*. The clauses of the CNF so far remain unchanged. We simply add new clauses derived from the new added dependency. The CNF takes a form $(t_{CrO,OD} \vee s_1) \wedge (\neg t_{CrO,OL} \vee s_1) \wedge (\neg t_{CrO,CA} \vee s_1) \wedge (\neg t_{CrO,OD} \vee s_2) \wedge (\neg t_{CrO,OL} \vee s_2)$. The CNF is unsatisfiable with the assumptions $\{s_1 = false, s_2 = false\}$ and the *HUMUS* returns $\{s_1 = false, s_2 = false\}$. So we isolate both dependencies. Obviously, the CNF becomes satisfiable with the new set of assumptions $\{s_1 = true, s_2 = ture\}$, but all affected cells remain empty as described above, so the matrix will be empty too.

As mentioned the isolated dependencies are kept isolated, and do not affect future input. Therefore if we add a new dependency we obtain some results from the trace analysis.

Step 3: Add *Dependency 3*. The CNF takes a form $(t_{CrO,OD} \vee s_1) \wedge (\neg t_{CrO,OL} \vee s_1) \wedge (\neg t_{CrO,CA} \vee s_1) \wedge (\neg t_{CrO,OD} \vee s_2) \wedge (\neg t_{CrO,OL} \vee s_2) \wedge (t_{CrO,OD} \vee t_{CrO,OL} \vee s_3) \wedge (\neg t_{CrO,CA} \vee s_3)$. One can see, that we work with the same CNF as in the case of the batch reasoning. The difference is that we preserve the isolation from the previous step by keeping the set of assumptions. That means we check the satisfiability for the set of assumptions $\{s_1 = true, s_2 = true, s_3 = false\}$. The CNF is satisfiable, and the oracle return the answers : $\{t_{CrO,OD} = true/false, t_{CrO,OL} = true/false, t_{CrO,CA} = false\}$. Table 4.6 shows the resulting matrix.

Table 4.6: Mobile Waiter trace matrix after step 3

	01: CrO	02: ClO
OrdersData(OD)		
Offline(OL)		
Calculator(CA)	N	

How we can see, we lose less information in case of incremental reasoning. However, results strictly depend on the order, in which dependencies are added. If, for example, the order would be like *Depenency* $1 \rightarrow$ *Dependency* $3 \rightarrow$ *Dependency* 2, a conflict would occur only on the third step, and, therefore, there were no difference with the batch reasoning.

Why do we need the support of incremental reasoning? If an engineer provides an incorrect dependency on some step, that affects many cells of the matrix, then in batch reasoning all dependencies directly and indirectly interrelated with the erroneous one will be isolated. In case of incremental reasoning, as this dependency has been isolated once it remain isolated until a user has corrected it and we potentially lose less data. Another advantage of this approach is that we are able to provide guidance about the fact, which of the isolated dependencies might be correct ones or incorrect ones based on newly provided information.

The idea is the following. If the last dependency in the list is not isolated, we assume it as a new data has been provided after last isolation. It means that we are dealing with a different CNF we had at the moment of the last isolation. In order to reason about the isolation, we

assume that the new input is correct; thus, we have additional information that is able to help to understand, which of the isolated dependencies are potentially correct.

For this purpose we divide the entire set of dependencies into two subsets: $Base$ and $Isolation$. Obviously these sets have no common elements and $Base \cup Isolation = Dependencies$. All dependencies from the set $Base$ we assume to be correct. For each dependency from the set $Isolation$ we perform the following steps:

1. Identify, whether the dependency to be checked has at least one common variable with at least one dependency from $Base$. If the answer is negative, we can not reason about this dependency as there are no common cells with the correct input. Such dependencies we call *yellow*.
2. If 1) gives a positive answer, we check whether the union of $Base$ with the dependency to be checked produce a conflict. If yes, such dependency seems to be incorrect (because $Base$ is assumed to be correct!). We call this dependency *red*. If not, such dependency seems to be correct, and we mark it as *green*.

As all isolated dependencies get colors, a user can begin to fix isolated dependencies in the order: $red \rightarrow yellow \rightarrow green$. This order minimizes the effort needed to bring the CNF in SAT-state comparing to the random correcting of dependencies or some other strategy.

This approach can be formalized as the algorithm "Coloring of dependencies". The additional methods we need are `getLastDependency()` returning the reference on the last added dependency, `getBase()` that returns a subset of dependencies containing only not isolated dependencies, `getIsolatedDependencies()` that returns the set of isolated dependencies and `constructCNF()` representing the algorithm for CNF construction. The method `addDependency()` adds clauses derived from the given dependency to the CNF and returns *true* if these clauses have at least one common variable with the clauses of the CNF and $false$ otherwise. If $false$, we do not need to check the satisfiability because the CNF is satisfiable (the base CNF is satisfiable by default, and the new dependency has no common variables, so there is no source for the conflict). This dependency gets the *yellow* color.

After the execution of the algorithm, each of the isolated dependencies has one of three colors that we use as guidelines.

We illustrate the algorithm on the example. Assume we have the input provided incrementally for the matrix presented in in Table 3.7:

```
Dependency 1:  {ClO} ImplExactly {OD}
Dependency 2:  {CrO} ImplAtLeast {OD, OL}
```

Algorithm 9 Coloring of dependencies

```
1: if not isIsolated(getLastDependency()) then
2:     baseCNF ← constuctCNF(getBase());
3:     for all dependency ∈ getIsolatedDependencies() do
4:         cnf ← baseCNF.clone();
5:         hasCommonVariables = cnf.addDepenendcy(dependency);
6:         if not hasCommonVariables then
7:             setColor(dependency, yellow)
8:         else
9:             if cnf.isSAT() then
10:                setColor(dependency, green)
11:            else
12:                setColor(dependency, red)
13:            end if
14:        end if
15:    end for
16: end if
```

`Dependency 3:` {`CrO, ClO`} `ImplNot` {`OD`}

`Dependency 4:` {`CrO`} `ImplNot` {`OD, CA`}

While adding the dependencies one by one, the conflict occurs after the addition of the *Dependency 3* and *HUMUS* isolated all three added dependencies. After addition of the last *Dependency 4*, the matrix takes the form showed in Table 4.7.

Table 4.7: Mobile Waiter trace matrix after step 4

	01: CrO	02: ClO
OrdersData(OD)	N	
Offline(OL)		
Calculator(CA)	N	

Now we have isolated dependencies, and we have not isolated tail of the dependencies chain. So we can apply the guidance algorithm. The set *Base* consists of only not isolated dependencies $Base = \{Dependency4\}$. As the *Dependency* 4 can be decomposed on two *NTRs*, the baseCNF will take the form: $baseCNF = (\neg t_{CrO,OD}) \wedge (\neg t_{CrO,CA})$. The next step is the assignment the color to each isolated dependency:

- `Dependency 1:` {`ClO`} `ImplExactly` {`OD`} produces the following clauses for the CNF: $\{(t_{ClO,OD}), (\neg t_{ClO,OL}), (\neg t_{ClO,CA})\}$. After adding these clauses to the *baseCN*F we get $cnf = (\neg t_{CrO,OD}) \wedge (\neg t_{CrO,CA}) \wedge (t_{ClO,OD}) \wedge (\neg t_{ClO,OL}) \wedge (\neg t_{ClO,CA})$. However we see, that the baseCNF has no common variables with clauses of the *Dependency 1*. So the *Dependency 1* is yellow.
- After adding the clauses of the dependency `Dependency 2:` {`CrO`} `ImplAtLeast` {`OD, OL`} to the *baseCNF* we get $cnf = (\neg t_{CrO,OD}) \wedge (\neg t_{CrO,CA}) \wedge (t_{CrO,OD}) \wedge (t_{CrO,OL}) \wedge$

($t_{ClO,OD} \vee t_{ClO,OL}$). The cnf is unsatisfiable because of the clauses one and three, so the dependency is red.

- After adding the clauses of the dependency `Dependency 3: {CrO, ClO} ImplNot {OD}` to the $baseCNF$ we get $cnf = (\neg t_{CrO,OD}) \wedge (\neg t_{CrO,CA}) \wedge (\neg t_{CrO,OD}) \wedge (\neg t_{ClO,OD})$. The cnf is satisfiable, so the *Dependency 3* is green.

The resulting guidance provided to the user based on the described analysis is represented in the Table 4.8.

Table 4.8: User guidance

Dependency	Color	Conclusion
Dependency 1	Yellow	–
Dependency 2	Red	Probabaly erroneous
Dependency 3	Green	Probabaly correct

The guidance is also available for other isolation strategy. The only difference is that we need to identify the set of isolated and no isolated clauses to establish $Base$ and $Isolated$ sets.

Chapter 5

Evaluation

Our approach designed to assist developers within the traceability capturing process uses SAT-based reasoning for the trace analysis. The purpose of the evaluation is to prove that it gives correct results according the semantic of dependencies. From the other hand, it is also necessary to show that the approach is efficient and is able to help the user to resolve uncertainties based on the provided dependencies.

Of course, the approach must be also useful for large systems, and the trace analysis must be performed in a reasonable time even if a user provides a large number of dependencies and hundreds of thousands artifact groups and no-trace relations.

To assess the isolation strategies for the traceability problem, we evaluated them for three third-party open source software systems of different sizes: *GanttProject*[1], *JHotDraw*[2] and *ReactOS*[3]. These systems were chosen because of the availability of high-quality RTMs for each of them, provided by the institute for Systems Engineering and Automation (SEA) of the Johannes Kepler University Linz. Because of high quality we can consider these RTM as the golden standard and use for results assessment.

GanttProject is a cross-platform desktop tool for project scheduling and management. It allows users to create project tasks, draw dependencies, define milestones, assign human resources to tasks etc. The system is large and contains more than 40,000 lines of code (LOC) written in Java, 500 classes and 2,500 methods. JHotDraw is a Java framework for drawing technical and structured Graphics. The system has a relative simple user interface, which allows to select, modify and draw different graphical elements. The project is implemented in about 71,000 LOC and about 2,000 methods. ReactOS is a free open-source operating system based on the Windows

[1] http://www.ganttproject.biz/
[2] http://www.jhotdraw.org/
[3] http://www.reactos.org/

XP/2003 architecture. The system has the purpose of replicating the Windows-NT architecture created by Microsoft on all the layers from the hardware to the application layer. It uses 23 different programming languages, like, for example, C, C++ and Assembler and has 5,237,905 LOC. The key characteristics of the mentioned systems like the number of requirements (#r), number of code artifacts (#ca), type of code artifacts (type), number of cells in golden RTM (#cells), and the number of SAT clauses generated per test case (#clauses) are presented in the Table 5.1.

TABLE 5.1: Characteristics of assessed systems

System	**#r**	**#ca**	**type**	**#cells**	**#clauses**
GanttProject	18	2,507	methods	45,126	0–150K
JHotDraw	21	1,610	methods	33,810	0–120K
GanttProject	16	239	classes	3,824	0–30K

Since it is hard to assess the approach using real life scenarios, we decided to perform the evaluation based on the random generated data. However, this data is not fully artificial as we used the real trace matrix of the real systems. We generated test cases using a special algorithm generating a set of dependencies according few predefined parameters based on the existing trace matrix.

5.1 Test cases generation

Engineers may provide dependencies of different types and different sizes. Different types of dependencies give different amount of certainties and uncertainties. For instance, single *ImplAtLeast* dependency results only in uncertainties if it contains more than one artifact from each side. Whereas an *ImplAtMost* dependency provides also certainties ($no - trace$ relations in most cases). To evaluate the efficiency taking into account different sizes and types of dependencies, we emphasis the relevance of the following parameters:

- *Percentage of types.*
- *Number of source artifacts and target artifacts in dependencies.* For instance, in the dependency $\{r\}\ ImplAtLeast\ \{c\}$ r is a source artifact, c is a target artifact. It may be also relevant to specify the percentage of the number of all possible artifacts. For instance, if we specify for source artifacts 25%, assuming requirements as source artifacts and having 20 requirements, number of source artifacts must be 5. This allows us to control the size of dependencies with the connection to the problem size.
- *Number of erroneous dependencies.* We have to be able to seed errors in generated test cases to evaluate the effectiveness of isolation strategies. This number allows us to control the faulty part of the input.

- *Errors distribution in dependencies.* As long as we decide to introduce an error in some dependency we have to understand what does it mean. We distinguish two types of errors: *false positives*(FP) and *false negatives*(FN). False positive errors include false traceability information, while actually this information does not exist. For example, the dependency $\{r1\}\ ImplAtLeast\ \{c1\}$ implies T in the cell $r1 - c1$. If we add some target artifact (e.g. $c2$), the cell $r1 - c2$ will also contain T. If $r1 - c2$ must not contain T, we introduced false positive error. The second type is false negative error, when we remove some artifacts from the dependency and lose information in the trace matrix. Error distribution allows to control the number of FP and FN errors inside of erroneous dependency.
- *Errors source.* We use for testing *Golden RTMs.* Cells of these RTMs are qualified as *trace* and $no - trace$, and we do not consider TN or empty cells here. The trace matrices filled by students for the same products are our errors source. These RTMs contain errors, and we inject these errors in the correct dependencies generated based on Golden RTMs. This method allows us to simulate real errors can be made by engineers while working on trace matrix.
- *Dependencies direction* which can be of two types. Source artifacts are requirements (therefore, target artifacts are elements of source code, direction $R \rightarrow C$), and source artifacts are elements of source code (direction $C \rightarrow R$).

For generating of test cases, we use an algorithm that create a set of dependencies based on the given trace matrix using predefined parameters describing above. The example of a parameters set may be the following:

- Number of generated dependencies: 100.
- Generate 25% of dependencies of each type (25 $ImplAtLeast$, 25 $ImplAtMost$, 25 $ImplExactly$, 25 $ImplNot$).
- Each dependency contains from 10 to 20 percent of source artifact. We have for the direction $R \rightarrow C$ from 2 to 4 requirements in each dependency if the overall number of requirements is 20. The number for each separate dependency is selected randomly.
- Each dependency contains 20 to 40 percen from basic dependency (see below).
- 10% of dependencies must contain errors.
- Number of FP and FN errors in each erroneous dependency must vary from 5% to 20%. That means that 5% to 20% of target artifacts must be deleted (FN) or added (FP).

The algorithm of generating test cases can be described as the following sequence of steps:

1. Identify all parameters of a test case: number of dependencies, errors distribution, etc.

2. For each dependency to be generated perform *step 3*.
3. Identify the type of the current dependency to be generated according the dependencies distribution and the number of already generated dependencies. Perform *step 4*.
4. Generate basic dependency. Basic dependency is always an $ImplExactly$ dependency as it contains the most information. According source artifact distribution, we select the corresponding number of random(but distinct) source artifacts from Golden RTM and include in the dependency target artifacts that have trace in Golden RTM to the selected source artifacts. Such selection guarantees that the generated basic $ImplExactly$ dependency will never contradict with information in the Golden RTM.
5. Transform basic dependency into dependency of the selected type identified on the *step 3*.
   - $ImplAtMost$: add the necessary number of random picked target artifacts to the target artifacts of basic dependency.
   - $ImplExactly$: do not change the basic dependency.
   - $ImplNot$: randomly select the necessary number of target artifacts from the set $AllTargetArtifacts - basicDependency.targetArtifacts$. According the semantic of $ImplExactly$ dependency the generated $ImplNot$ dependency will provide $no-trace$ relations for the cells of trace matrix not containing T in the Golden RTM (they are included in the basic dependency) and will never contribute a contradiction.
   - $ImplAtLeast$: see below.

   After this perform the *step 6*.
6. Inject the necessary number of errors in the dependency.

Now we present two algorithms of error injection (algorithm 10) and transformation of basic dependency to the *ImplAtLeast* dependency used in the main algorithm of test case generation.

For the injection of student errors we assume that we have N_{st} trace matrices provided by N_{st} students. These matrices contain errors (relation in cell differing from the relation in the same cell of Goldern RTM), and we inject these errors when needed.

While straight forwarf transforming basic $ImplExactly$ dependency to an $ImplAtLeast$ dependency the following situation is possible. Assume, we have a basic dependency $\{r_1, r_2, r_3, r_4\}$ $ImplExactly$ $\{c_1, c_2, c_3, c_4, c_5\}$ for the corresponding excerpt of the Golden RTM:

Algorithm 10 Injection of errors in dependency

Input:
- number of FP and FN artifacts (fp, fn))
- basic dependency (*basic*)
- original dependency (*original*)

Output : set of target artifacts for dependency with errors

1: $targetsComplement \leftarrow allTargets - basic.targets$ ▷ calculate all targets not in basic dep.
2: **if** original.type not "NOT" **then**
3: $artifactsFN \leftarrow getStudentErrors($
$submatrix : basic.sources \times basic.targets$
$errorsToSearch : FN)$;
4: $artifactsFP \leftarrow getStudentErrors($
$submatrix : basic.sources \times targetsComplement$
$errorsToSearch : FP)$;
5: **else** ▷ ImplNot-dependency
6: $artifactsFN \leftarrow \oslash$;
7: $artifactsFP \leftarrow getStudentErrors($
$submatrix : basic.sources \times basic.targets$
$errorsToSearch : FN)$;
8: **end if**
9: $artifactsFN \leftarrow randomSubset(artifactsFN, fn)$;
10: $artifactsFP \leftarrow randomSubset(artifactsFP, fp)$;
11: $targets \leftarrow original.targets - targetsFN$;
12: $targets \leftarrow targets + artifactsFP$;
13: **return** $targets$;

	r_1	r_2	r_3	r_4
c_1	×			
c_2	×	×		
c_3		×	×	
c_4			×	×
c_5				×

To consruct an *ImplAtLeast* dependency with 2 target artifacts we randomly select two artifacts of basic dependency (for example, c_1 and c_2). The resulting dependency will have the form $\{r_1, r_2, r_3, r_4\}$ *ImplAtLeast* $\{c_1, c_2\}$ that implies the existence of the artifact group $AG(r_3, \{c_1, c_2\})$. As there are no traces in cells $r_3 - c_1$ and $r_3 - c_2$, we have a conflict. To avoid this, we have to select two targets in the way, that no conflicts occur. Representing the matrix in form of Boolean strings we get the following:

$c_1 \rightarrow 1000$

$c_2 \rightarrow 1100$

$c_3 \rightarrow 0110$

$c_4 \rightarrow 0011$

$c_5 \rightarrow 0001$

Selected targets for $ImplAtLeast$ c_x and c_y must return Boolean string without $0s$ in case of applying bitwise OR.

$c_1\ OR\ c_2 = 1100$. There are $0s$, conflict.

$c_2\ OR\ c_4 = 1111$. There are no $0s$, no conflict.

In order to select targets by this way, one can first calculate the basis (minimal set of targets not causing a conflict):

1. Sort targets by cardinality : $basis = \{c_1, c_5, c_2, c_3, c_4\}$.
2. Step by step remove one target and check the consistency.
   (a) remove first artifact c_1, $basis = \{c_5, c_2, c_3, c_4\}$: $c_5\ OR\ c_2\ OR\ c_3\ OR\ c_4 = 1111$.
   (b) remove first artifact c_5, $basis = \{c_2, c_3, c_4\}$: $c_2\ OR\ c_3\ OR\ c_4 = 1111$.
   (c) remove first artifact c_2, $basis = \{c_3, c_4\}$: $c_3\ OR\ c_4 = 0111$. Conflict, continue with $basis = \{c_2, c_3, c_4\}$
   (d) remove second artifact c_3, $basis = \{c_2, c_4\}$: $c_2\ OR\ c_4 = 1111$.
   (e) remove second artifact c_4, $basis = \{c_2\}$: $c_2 = 1100$. Conflict, continue with $basis = \{c_2, c_4\}$
   (f) remove third artifact. $|basis| = 2 \rightarrow$ end.

Minimal basis for $ImplAtLeast$ is the set $\{c_2, c_4\}$. These artifacts have to be included into the dependency. Rest artifacts can be added randomly.

5.2 Test cases parameters

For the evaluation, we created test cases with different combinations of parameters. Configurations with dependency distributions: 25% of each type; 40% of one type, 20% of each other type (overall 4 configuration); 55% of one type, 15% of each other type. Each test case contains 30 dependencies with direction $R \rightarrow C$. For each of these configurations: one test case in which the number of source artifacts is 1 and number of target artifacts is 50% (from the number of all target artifacts); one test case in which number of source artifacts is 3, where the range of target artifacts is 75%. Each test case contains one erroneous dependency with one FP or FN artifact(we select randomly one of them).

Each combination of described parameters gives one configuration. For each such configuration, we generated 6 test cases. We also preserved a test case, in which the error is not presented, so at the end we have additional 6 error free test cases for each configuration.

Additionally, we generated 5 test cases for the configuration with the dependencies distributions: 70% of one type, 10% of each other type; 100% of each type. Other parameters remain the same except errors. We do not inject them in these test cases.

Resulting, we have 108 test cases with errors and 188 error free test cases with 30 dependencies each. To test the correctness and scalability for larger problems we also generated test cases with 100 and 500 dependencies. Each configuration contains 25% dependencies of each type, but we vary the error seeding range from 0% to 20% with step 10% for FP and FN, that gives 9 different configurations. For each of these configurations, we generate one test case in which number of source artifacts is 3, where range of target artifacts is 75%. Here, we have 9 error free test cases and 8 erroneous test cases.

We reduced the number of test cases for larger problems because they suffice to assess the scalability and behavior in case of a big amount of errors, whereas we generated for smaller problems 300 test cases varying all parameters that give us statistically reliable results.

The same configurations we used to generate test cases for Gannt, JHotDraw and ReactOS systems. Overall number of generated test cases is 990.

5.3 Correctness and efficiency

Dependencies are generated based on their semantics, and we do not use the reasoning mechanisms for this. If we do not seed errors in a test case, reasoning must produce result not contradicting with the Golden RTM. In other words, if some cell of the Golden RTM contains T, the corresponding cell of the resulting matrix may contain T, TN or be empty (the same for N).

All experiments show that the reasoning is always correct, so the method itself does not cause conflicts. Another question is the effectiveness of the approach. We need to determine how results cover the trace matrix. In other words, we have to understand the percentage of correct derived Ts and Ns in the result. Of course, some input may produce matrices with only empty cells, but as we generate test cases randomly including all types of the dependency, the probability of this outcome is negligible.

While testing we performed the emulation of incremental reasoning as it is also essential to understand the impact of each added dependency on the result.

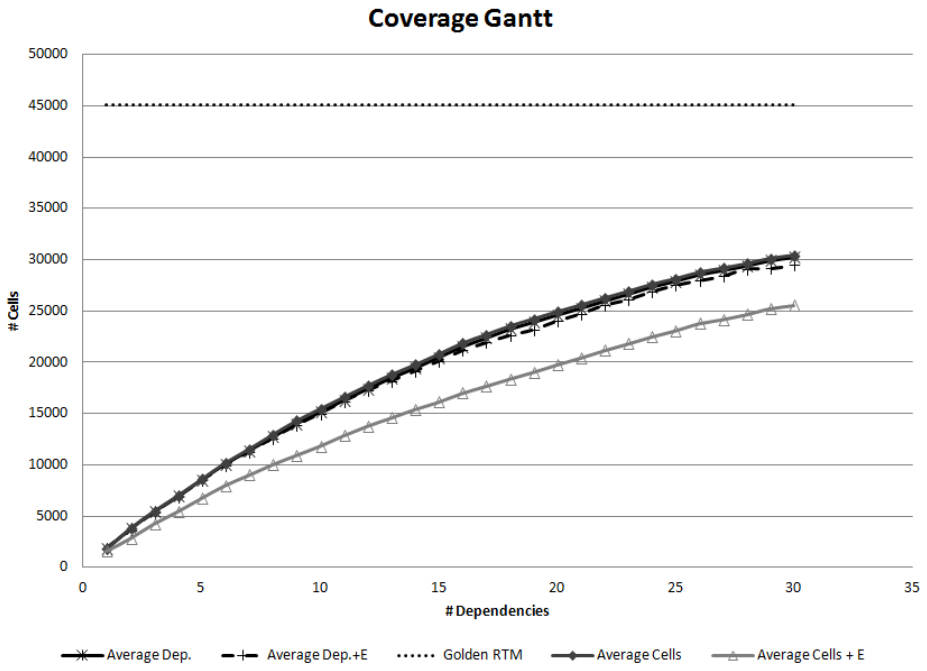

FIGURE 5.1: Coverage for GanttProject

Very similar pictures we can observe for ReactOS and JHotDraw. So we can see, after each step the coverage increases and tends to the maximal possible value (in Golden RTM). The diagram demonstrates that the coverage for test cases with errors remains behind the coverage for error free test cases. The reason is the isolation performed by HUMUS that causes the unavoidable loss of information. Relative fast converging to the bound may be explained as the result of artificial nature of test cases. They were generated randomly, and the size of dependencies is large, what is difficult to produce in real live. For real tasks, it is also impossible to measure the coverage as there is no golden RTM, but intuition suggests that the coverage growth is much slower. In any case, the observed behavior corresponds the expected behavior.

5.4 Isolation

We showed that the technique itself demonstrates correct behavior for error free test cases. What is also significant is to understand how effective is the isolation, how much information we lose while isolation, whether the erroneous dependencies are always identified.

We performed testing for two isolation levels: dependencies and cells levels. Moreover, when we injected errors, we marked the corresponding dependencies as erroneous, what makes no impact on the reasoning, but allows to identify, whether the actual faulty dependencies are found and isolated by HUMUS. The following diagrams demonstrates the proportion of the number of

isolated dependencies to the number of all dependencies in the test case, what shows the indirect loss of information. Clear, the smaller the number (the less dependencies are isolated), the less information we lose. First picture represents information for test cases with 30 dependencies and one error, the second for test cases with 100 and 500 dependencies and the errors rate from 10 to 40% (of all dependencies) for isolation on cells level.

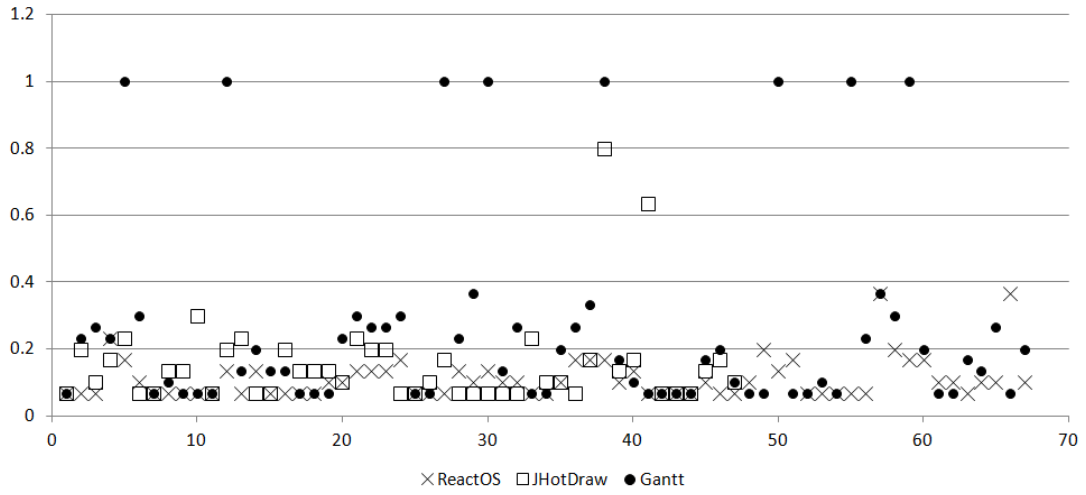

FIGURE 5.2: Isolated dependencies vs. all dependencies

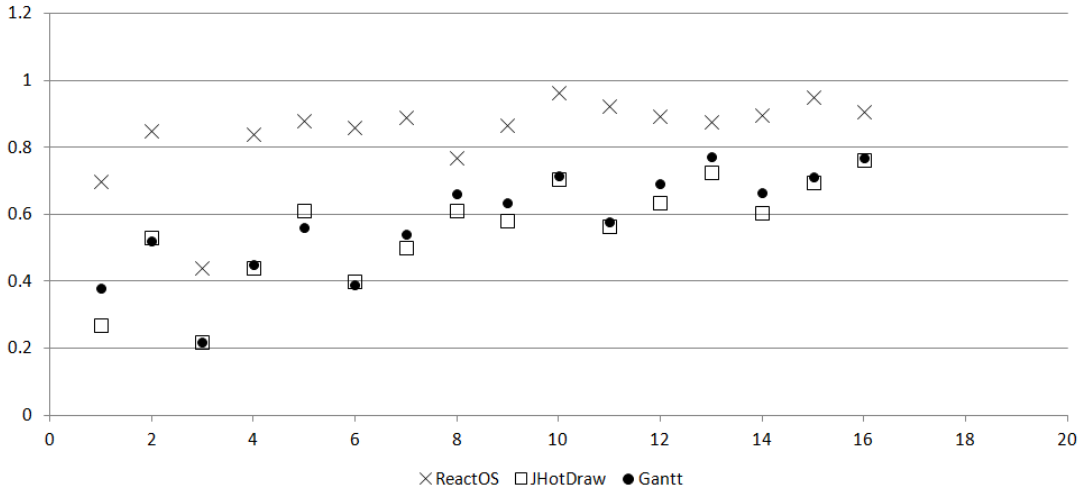

FIGURE 5.3: Isolated dependencies vs. all dependencies (cells level)

As one can see from the first diagram, for 70% of test cases with one error the percentage of isolated dependencies does not exceed 20, for the rest 30% does not exceed 40. The results do not contradict expectation, as for random generated test cases many dependencies are interrelated, and HUMUS isolates all direct and indirect conflict contributors. For a large number of errors the ratio varies from 20 to 100 percent, and one can conclude that the best strategy is to fix the problem after its identification to minimize data loss.

The following illustration demonstrates the direct loss of data. In other words, how much information (coverage) we lose in average working with error free and erroneous test cases while incremental reasoning. This is built based on the test cases without errors and the same test cases, but with one erroneous dependency(one FP or FN error).

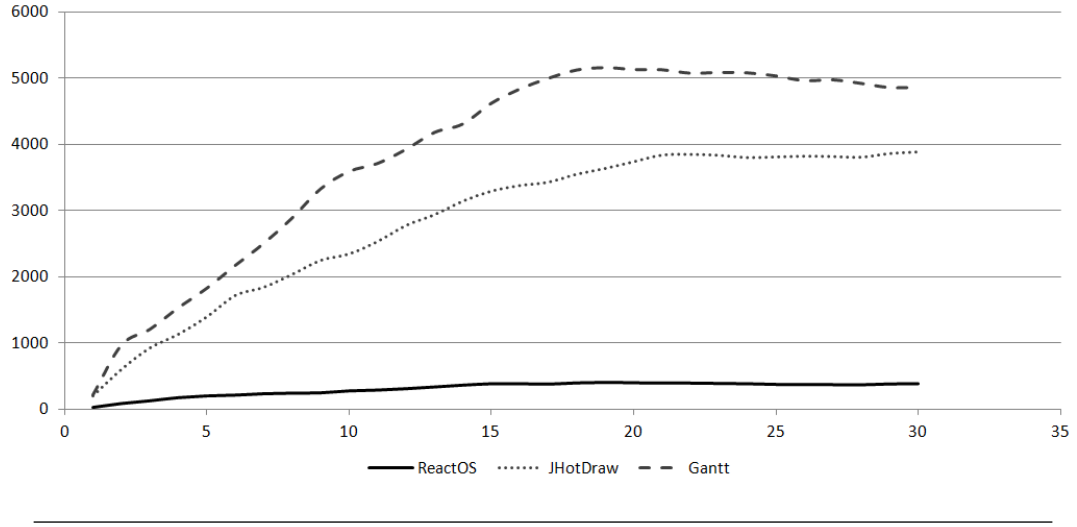

FIGURE 5.4: Different of coverages (cells level)

We see that in average such data loss increases but does not exceed some limit determined by dependencies involved into isolation.

5.4.1 Recall

Under recall we understand the number of identified erroneous dependencies divided by the number of all erroneous dependencies. Recall shows how effective the error identification.

For test cases with one faulty dependency recall is 100%, what shows that HUMUS always identifies single defect. For multiple seeded errors not all of them are always identified, because they may still produce consistent input. Another reason is that we performed testing with incremental reasoning, so that some erroneous dependencies could not be identified as defects because of isolated dependencies on previous steps.

As we see from Figure 5.5, recall is still high for large test cases recall, what demonstrates the high efficiency of HUMUS in traceability analysis.

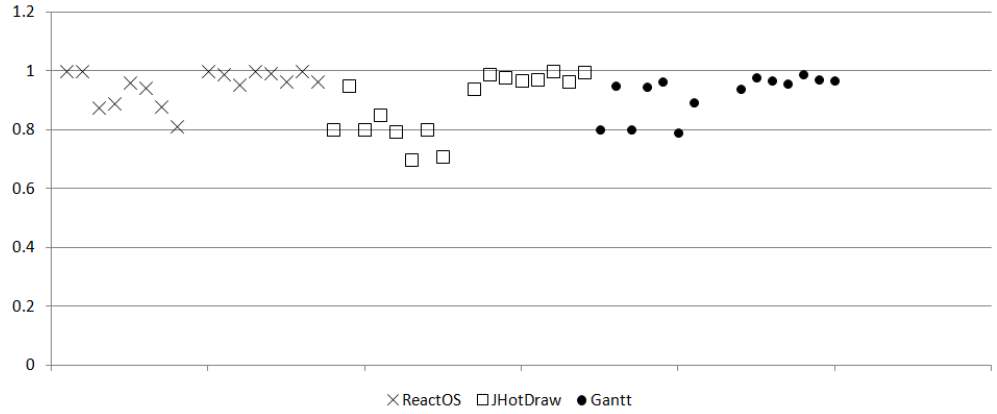

FIGURE 5.5: Recall

5.4.2 Precision

Precision is the number of isolated erroneous dependencies divided by the number of all isolated dependencies. Precision shows how much additional information (that does not belong to the conflict) is isolated. The lower is precision, the more information we lose.

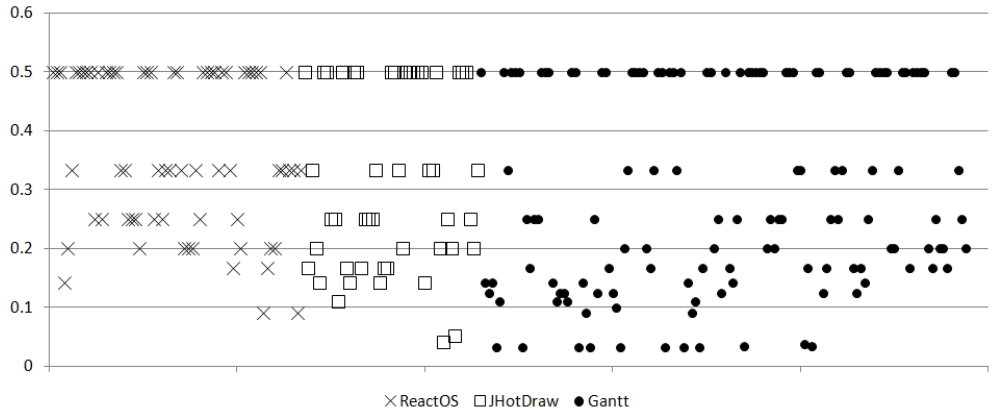

FIGURE 5.6: Precision

In case of one error precision does not exceed 0.5, that is upper bound because HUMUS can not isolate single dependency as the erroneous dependency must contradict at least one correct dependency to be isolated. However, for most cases, precision varies from 0.1 to 0.3, and the information loss is relative high. This can be explained by the nature of HUMUS and randomly generated dependencies. For larger test cases, precision is higher but still beyond the ideal values.

We computed recall and precision for test cases with 30, 100 and 500 dependencies with isolation on the dependency level and 100 and 500 dependencies with isolation on cell level. Of course, even if we seeded errors in the input, it does not guarantee that this error will be identified if the input will be consistent (CNF is satisfiable). The ratio of test cases where the error has been

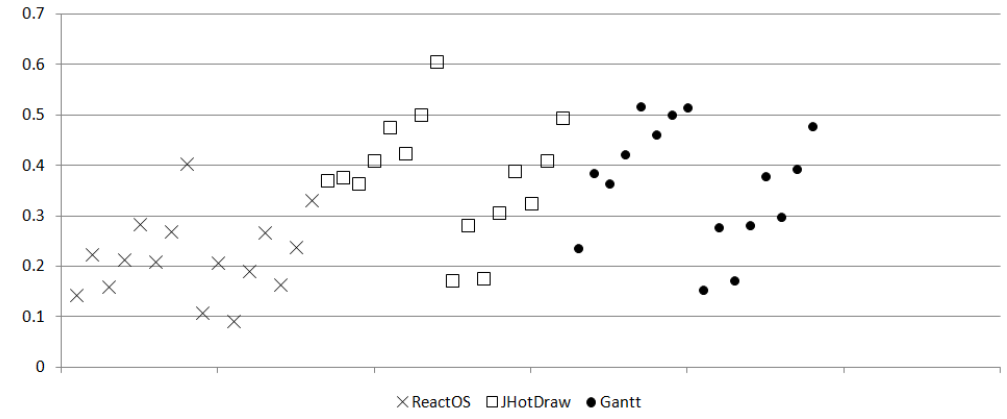

FIGURE 5.7: Precision (cells level)

identified to the number of test cases where the error has been injected is not relevant because this number is random and not related to the approach.

5.5 Scalability

Another important ability of the approach is to perform the reasoning for large input in a reasonable time.

The following diagram demonstrates the execution times for different problem sizes for JHotDraw system. The problem size is the number of units (artifact groups and no-trace relation) derived from the provided dependencies. This number is equal to the number of clauses in CNF.

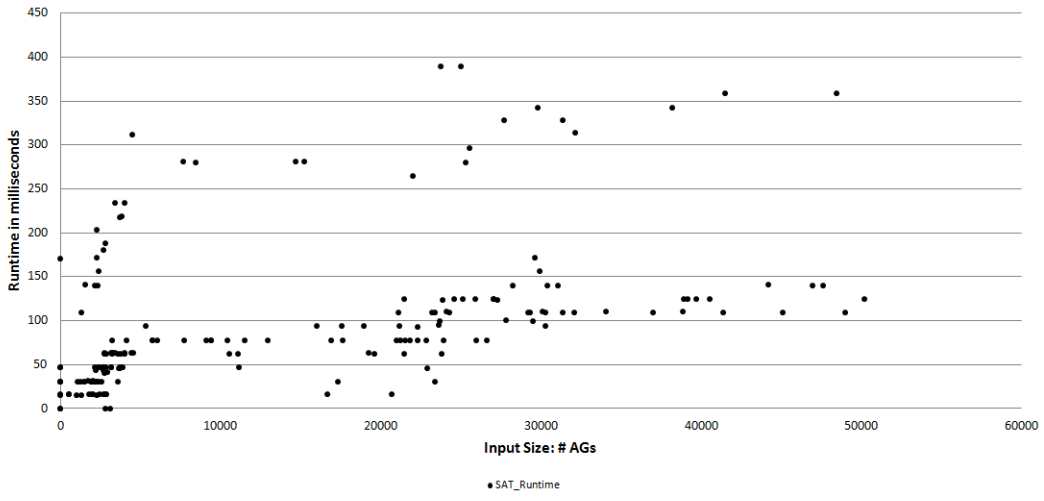

FIGURE 5.8: Scalability (small problems)

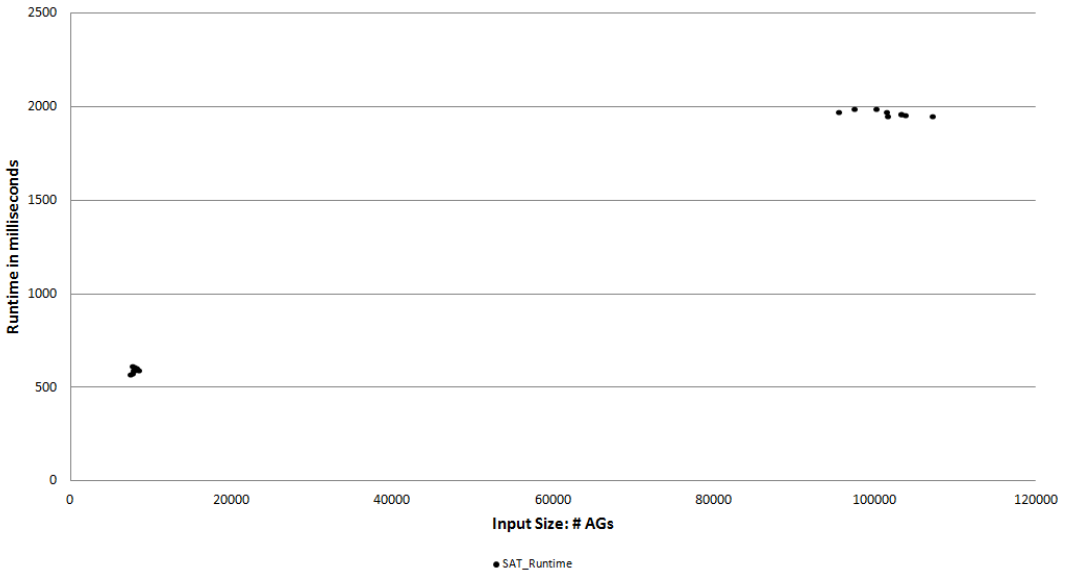

FIGURE 5.9: Scalability (large problems)

Of course, HUMUS requires additional time, and in case of batch reasoning the overall execution time is higher. Figure 5.10 demonstrates execution time in case of isolation on cell level.

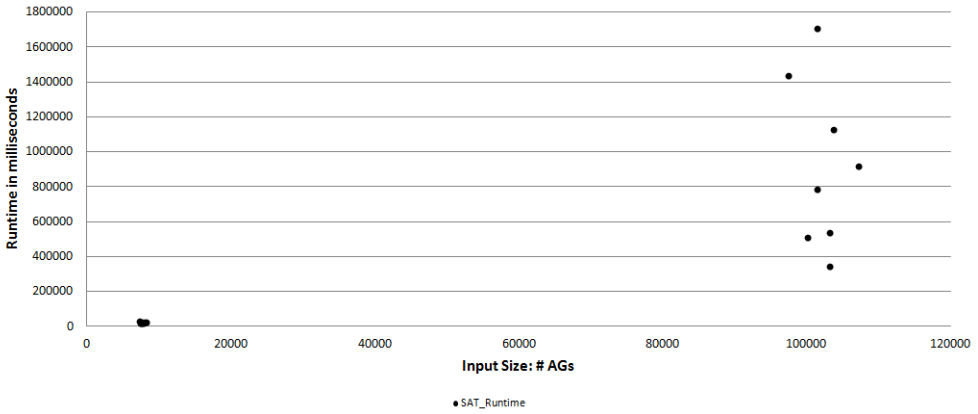

FIGURE 5.10: Scalability (with errors)

In case of incremental reasoning, (first 200 dependencies) the time of the analysis increases with the number of dependencies. However, the difference between reasoning with errors and without them is minimal.

We also performed the evaluation for test cases with dependencies in both direction $R \rightarrow C$ and $C \rightarrow R$ (30 dependencies per direction) and obtained similar results. Therefore, we conclude that the direction specify the semantic of dependencies (and, thus, derived artifact groups and no-trace relation), but do not influence the reasoning and isolation.

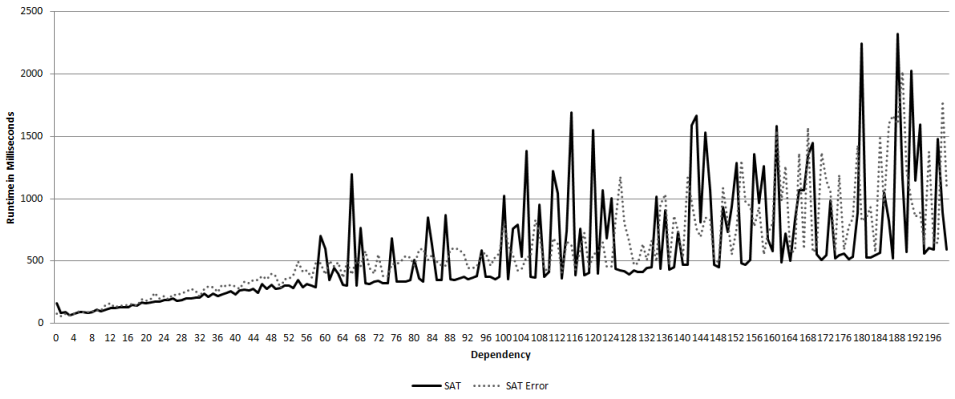

FIGURE 5.11: Scalability (incremental reasoning)

Chapter 6

Tool

Our approach is automatic but requires specific input data (dependencies). Dependencies may be relevant for specific trace matrix. Description of the trace matrix must be also a part of the input. In this chapter we introduce TraceAnalyser - the tool supporting our method, its user interface and functionality.

6.1 Eclipse platform

TraceAnalyzer is not a standalone application. It is implemented as an Eclipse plug-in. Eclipse is an open source IDE(Integrated development environment) that provides a generic extensible plug-in system allowing to customize the environment. One of the advantages of Eclipse platform is that it is the free and open source software released under EPL(Eclipse Public License). The following reasons prompted the decision to use Eclipse as the platform:

- Eclipse is a cross-platform system, thus, once developed, we can use the tool on every machine with installed Eclipse and necessary plug-ins without adaptation.
- Eclipse provides the Rich Client Platform(RCP) for developing general purpose applications. For the implementation of UI components we used both Standard Widget Toolkit(SWT) and JFace that provides helper classes making development of some UI features(like tables) easier.
- Eclipse provides multiple views for the same model (like source code, outline, etc), what makes the implementation of MVC concept easier.
- There are already many free open source project for Eclipse that can be efficiently reused. For example, we used for the tool *KTable*[1] to display and operate with trace matrix.

[1] http://sourceforge.net/projects/ktable

KTable is a custom SWT table widget providing a flexible grid of cells to display data. Since it is custom-drawn, it does not have the restrictions of the native SWT Table control. We also used *Zest*[2] library for the visualization of footprint graph.

Eclipse provides an EMF (Eclipse Modelling Framework) and code generation facility for building tools and other applications we used for developing the data structure for representing dependencies, artifact groups, trace and no-trace relations, as well as trace matrix. Java classes are generated automatically from the model, and the code consistency is also controlled automatically if we change the model.

6.2 Tool architecture

TraceAnalyzer implements approaches described in chapter 3 and chapter 4 (trace analysis using SAT-based reasoning, isolation of erroneous dependencies, user guidance). The user input (set of dependencies, artifacts of trace matrix) stored as an EMF model but separated from the reasoning, isolation and guidance process. Isolated dependencies or units stored in separate data structure that allows to use other reasoning approach with possibly different isolation strategy to the same model.

The core structure of the application is presented on Figure 6.1

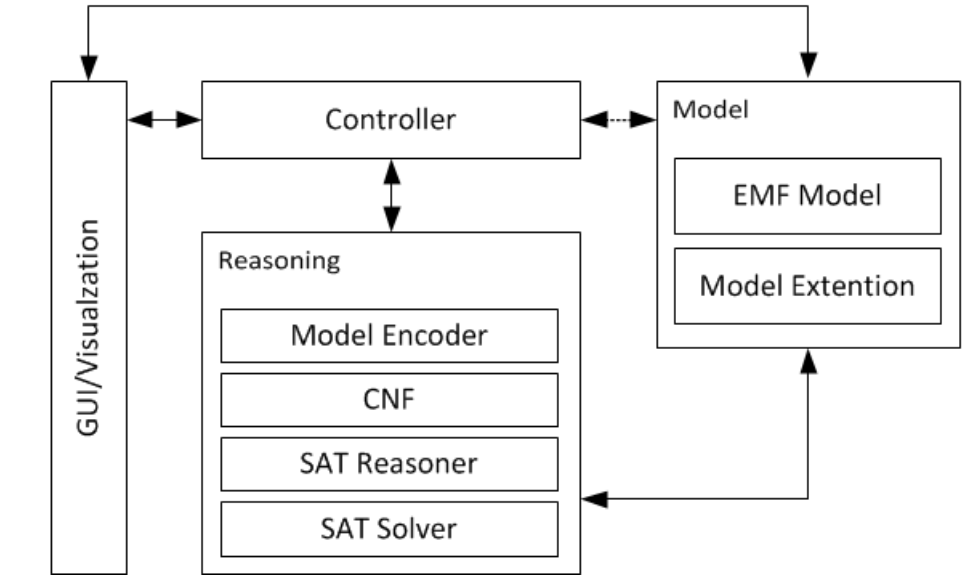

FIGURE 6.1: Tool structure

The Figure 6.2 demonstrates the class diagram for the reasoning subsystem of the tool.

[2]http://www.eclipse.org/gef/zest

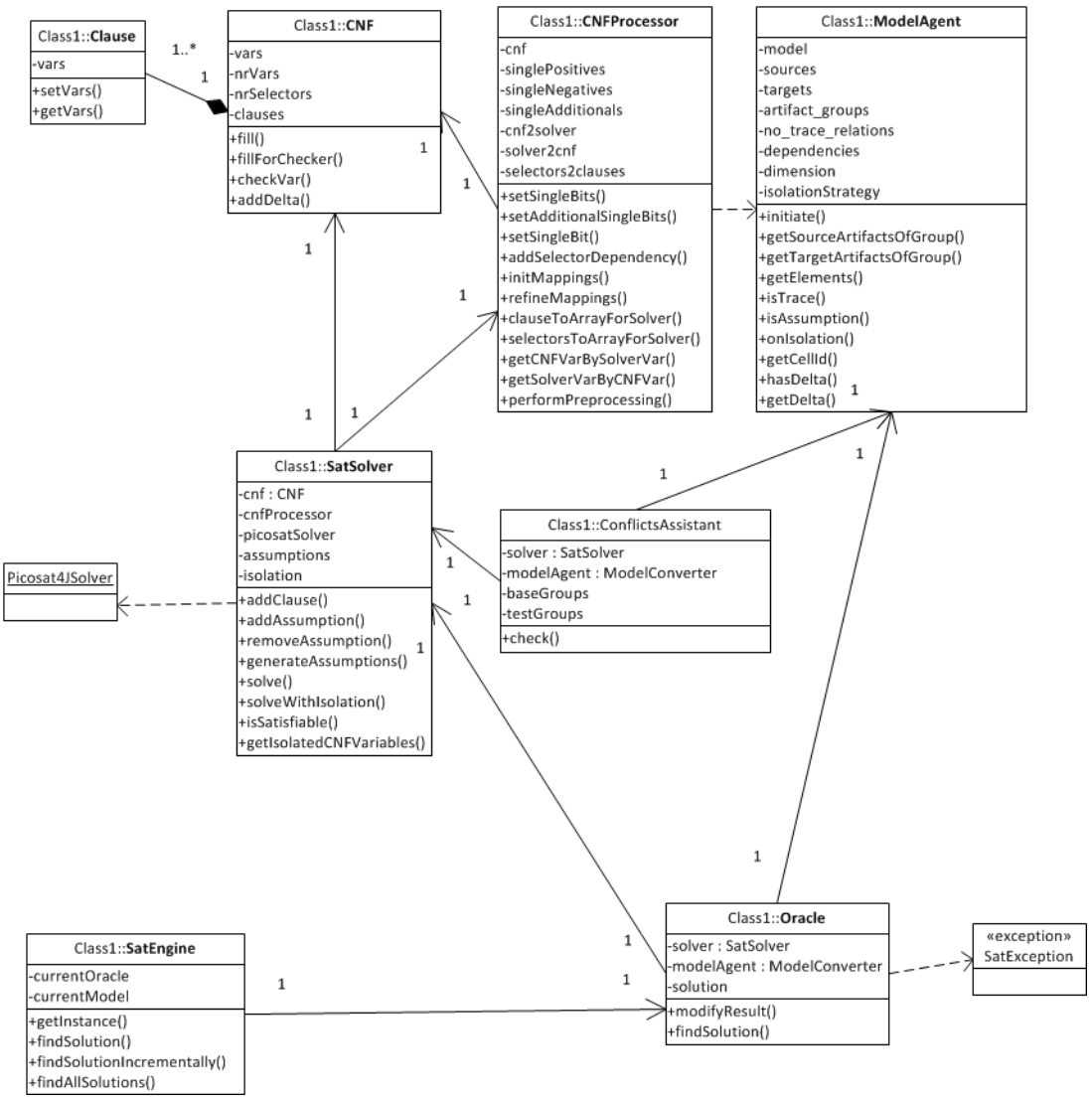

FIGURE 6.2: Class diagram

6.3 User interface

The user interface may be divided into four main categories: source code of problem description, model editor, model visualization and user guidance.

6.3.1 Source code of problem description

The source code is displayed in a view in the middle of workspace. The view in this case is a simple text editor without capabilities of syntax coloring or content assistance. The source code has a simple structure and the following core components:

- *Perspectives* describing types of artifacts for the traceability matrix (one matrix connects two perspectives; for example, requirements and source code) trough the specifying of perspective name and the list of artifacts within it.
- *Dependencies.* The order of dependencies in source code determines the order in the reasoning, what is essential for emulation of incremental reasoning as well as for user guidance.

The example of source code is the following:

```
perspective requirements {CrO, ClO, CSR, ChP, BgU};
perspective codes {Values, Action, Request, Utils, Order};

dependency {ClO, CrO} atLeast {Action, Values};
dependency {ChP} atMost {Action, Request};
dependency {ClO} exactly {Values, Utils, Order};
dependency {CSR} not {Request, Action};
```

6.3.2 Model editor

Model editor is implemented similar to the package explorer and organized as a tree. Each tree elements has a pop-up menu activated by the click of the right mouse button that provides available actions on this tree element. The tree has two root elements *Perspectives* and *Dependencies*. The following operations are available:

- For the root element *Perspectives*: expand/collapse subtree, add new perspective.
- For a single perspective: add artifact, rename perspective, remove perspective.
- For an artifact: rename artifact, remove artifact.
- For the root element *Dependencies*: add dependency.
- For a single dependency : edit dependency, remove dependency.

The form for the dependency editing (and dependency adding) allows to select source and target perspective from the specified perspectives, as well as source and target artifacts and the dependency type.

After each action in the editor all changes are applied to the source code and the reasoning performed, so every manipulation has an direct effect.

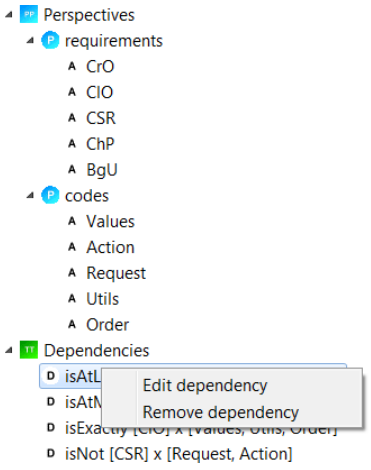

FIGURE 6.3: Model Editor

6.3.3 Visualization

Consist of trace matrix view, footprint graph view and outline view.

Trace matrix visualizes the trace matrix after reasoning. Artifacts of trace matrix are described in the source code; the cells of the trace matrix are the result of the reasoning. They may contain T(trace), N(no-trace), C(conflict, for isolation on cells level) or be TN(uncertainty or no-data, the same as empty). Figure 6.4 demonstrates an example.

sources\targets	Values	Action	Request	Utils	Order
CrO	TN	T	TN	TN	TN
ClO	T	N	N	T	T
CSR	TN	N	C	TN	TN
ChP	N	TN	T	N	N
BgU	TN	TN	TN	TN	TN

FIGURE 6.4: Trace matrix

Trace matrix serves also as input tool and allows to add dependencies by selecting cells and specifying the dependency type. Involved source and target artifacts are added to the new dependency; after addition the tool performs the reasoning and updates the matrix.

Footprint graph view depicts the footprint graph derived from the user's input (model) after reasoning. Its structure corresponds the structure described in the chapter 3; additionally, we show all trace and no-trace relation derived during the trace analysis. The example of such a graph is shown on the Figure 6.5.

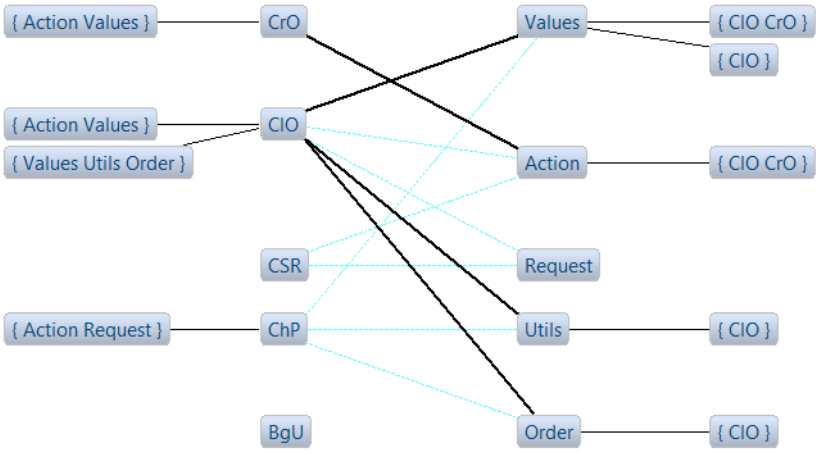

FIGURE 6.5: Footprint graph

Outline (example on Figure 6.6) provides information about derived artifact groups and no trace relations as well as isolated elements. In case of isolation on the dependencies level all isolated dependencies and their children are emphasized with red background color. If the reasoning works on units level, only involved units are emphasized with red background color. In order to designate dependencies that are parents of the isolated units, their text colors becomes red. The same visualization scheme we use for the isolation on the cells level; additionally cells of the matrix containing C are also emphasized with red background.

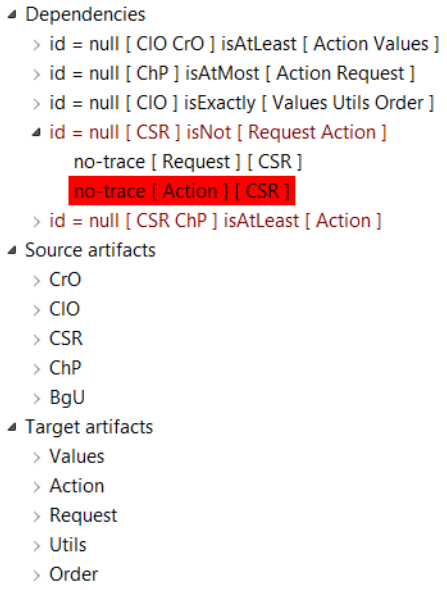

FIGURE 6.6: Outline

Outline provides also a possibility to remove selected dependency. For this purpose the user must select dependencies to be removed and select "delete" item in the pop-up menu. After this

operation the tool performs reasoning and all changes are immediately visible.

Selected elements are also emphasized in the trace matrix and footprint graph. In the trace matrix, affected cells become blue background; corresponding nodes and connection in footprint graph are emphasized by bold lines and yellow color.

6.3.4 User guidance

User guidance view visualizes the results of user guidance algorithm described in the chapter 4. It represents the list of isolated dependencies with the color emphasizing. The color meanings correspond the colors of the dependencies in the approach.

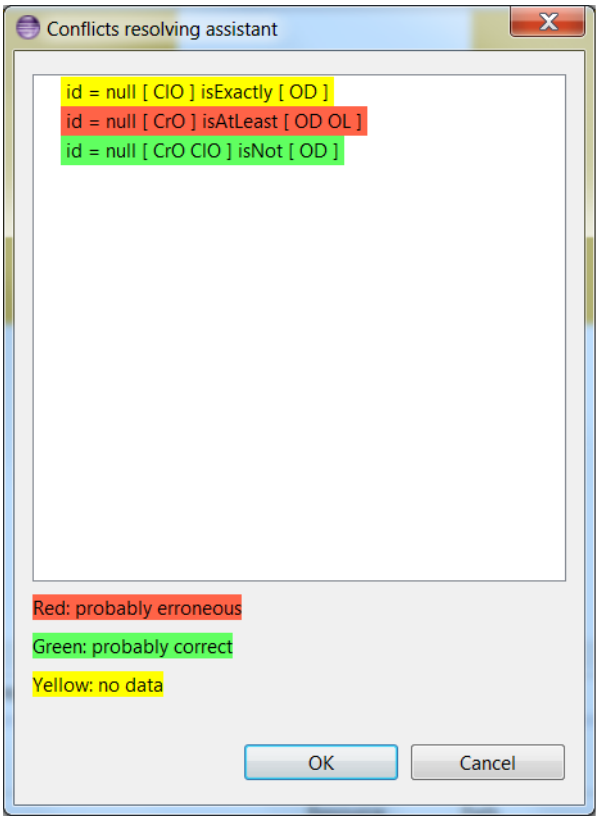

FIGURE 6.7: Conflict resolving assistant

6.4 Third party libraries

We used for the tool the following third party libraries:

- **KTable** for the visualization of trace matrix(see above).

- **Zest** for the visualization of footprint graph. Zest is the Eclipse Visualization Toolkit, is a set of visualization components built for Eclipse. The entire Zest library has been developed in SWT/Draw2D and integrates seamlessly within Eclipse because of its recognized design. Zest has been modeled after JFace, and all the Zest views conform to the same standards and conventions as existing Eclipse views. This means that the providers, actions and listeners used within existing applications can be leveraged within Zest. The Zest project also contains a graph layout package which can be used independently. The graph layout package can be used within existing Java applications (SWT or AWT) to provide layout locations for a set of entities and relationships.
- **ANTLR** for parsing of problem description source code into the EMF Model. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees. The grammar for our tool see in the appendix A.
- PicoSAT as Sat-solver.

Chapter 7

Conclusions And Future Work

7.1 Summary

This section presents the conclusions we draw from this thesis, including its results and the most challenging aspects.

The second chapter is the introduction in the field, where we briefly described traceability in order to better understand its purposes and capabilities. We showed that one can divide trace relations in different categories that have different goals and also mentioned several most important application areas of the traceability. We also discussed there different methods of traceability capturing, maintenance and recording.

In the third chapter we showed that manual traces capturing is error prone and complex process; therefore, it is more suitable to use automated approaches. We described related works in the traceability branch that motivated us to develop a new approach to the trace analysis using SAT-based reasoning.

During the thesis we provided the theoretical basis of the trace analysis that allows to represent user input in conjunctive normal form and, thus, use SAT-solvers as a core of the algorithm. We showed the unavoidability of conflicts in the user input and described different strategies to the conflicts resolution. It was decided to use the tolerating strategy as most appropriate for our problem and its implementation *HUMUS* because it supported by the PicoSAT solver and directly applicable to our problem.

The approach support the isolation of input elements on different granularity levels what allows comfortable work to users. Our algorithms are appropriate for both batch and incremental reasoning.

As the concept proof, we performed a comprehensive testing of our approach based on about 1000 random generated test cases, which were produced based on the existing trace matrices for real software systems. These test cases contain different numbers of dependencies of each type, different number of erroneous dependencies and different distributions of faulty artifacts within each dependency.

We showed that the basic characteristics of the algorithms like scalability, correctness, efficiency, isolation recall and precision are satisfactory, and the approach may be used for real problems.

While the focus of this thesis was primarily on the technologies, it also presented a possibility of usage and visualization of these technologies in a prototype tool, in which all aspects of the approach were involved. The tool itself is not useful in the industry, but we believe that we feel right direction and the prototype can be properly extended.

7.2 Future works

In this section we briefly provide our vision about what can be improved in our approach and which further steps can be done in this field.

7.2.1 Multidimensional reasoning

Our approach based on the trace matrix and its cells, which we represent through propositional variables. As the trace matrix connects only two dimensions(two types of artifacts), we can apply our algorithm without modifications to the trace analysis only between these two dimensions. Theoretically, one can modify the approach to support multi-dimensional reasoning and support multi-dimensional matrices. For example, an engineer may provide dependencies between requirements and test cases and test cases and source code. Based on this information we could extract direct trace relations between requirements and source code. Another possibility is to allow single dependency to connect artifacts of more than two dimensions. Multi-dimensional traceability matrices may be useful for implementing, testing and maintenance of complex and large-scale software systems.

7.2.2 User guidance for units and cells

The implemented user guidance provides information about possibly erroneous dependencies. One of the visible extensions is to make(similar to the isolation levels) guidance about possibly erroneous artifact groups, no-trace relations and cells. Such information may help the user

to understand an error better. Moreover, based on multi-dimensional reasoning one can even validate the consistence ot the input and propose possible corrections if needed.

Appendix A

ANTLR Grammar

```
grammar Traceability;

tokens {
PERSPECTIVE = 'perspective';
IMPORT = 'import';
REL = 'dependency';
SOURCE = 'source';
TARGET = 'target';
USE = 'use';
AS = 'as';
END = ';';
COMMA = ',';
ID_WORD = 'id';
IMPLATLEAST = 'atLeast';
IMPLATMOST = 'atMost';
IMPLEXACTLY = 'exactly';
IMPLNOT = 'not';
}

program
:  (perspective_declaration {$parserInput.addPerspective($perspective_declaration.value);})*
(relationship {$parserInput.addRelationship($relationship.value, $relationship.text);}
)* EOF
;
```

```
import_declaration
:  IMPORT id=ID END
;

perspective_declaration
:  PERSPECTIVE pname = ID '{' p1=ID (COMMA p2=ID)* '}' END
;

relationship
:  REL (ID_WORD '=' idvalue=ID )? '{' id3=ID } (COMMA id4=ID )* '}'
rel '{' id1=ID (COMMA id2=ID)* '}' END
;

rel
:  IMPLATLEAST |IMPLATMOST |IMPLEXACTLY |IMPLNOT
;

ID : ('a'..'z'|'A'..'Z'|'0'..'9'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'.')*
;

COMMENT
: '//'  ('\n'|'\r')* '\r'?  '\n' {$channel=HIDDEN;}
|'/*' ( options {greedy=false;} :  .  )* '*/' {$channel=HIDDEN;}
;

WS : ( ' '
|'\t'
|'\r'
|'\n'
|'\r \n'
) {$channel=HIDDEN;}
;

STRING
: '"' ( ESC_SEQ | ('\\'|'"') )* '"'
;

CHAR : '\'' ( ESC_SEQ | ('\''|'\\') ) '\''
```

```
;

fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
: '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
|UNICODE_ESC
|OCTAL_ESC
;

fragment
OCTAL_ESC
: '\\' ('0'..'3') ('0'..'7') ('0'..'7')
|'\\' ('0'..'7') ('0'..'7')
|'\\' ('0'..'7')
;

fragment
UNICODE_ESC
: '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
;
```

Bibliography

[1] The grand challenge of traceability (v1.0). *Center of Excellence for Software Traceability, Technical Report #CoEST-2011-001*, June 2011.

[2] O. Gotel and A. Finkelstein. Contribution structures (requirements artifacts). *Proceedings of 2nd International Symposium on Requirements Engineering, RE*, pages 100–107, 1995.

[3] *Handbook of Softawre Engineering and Knowledge Engineering*, volume 3, chapter Software traceability: A roadmap. World Scientific Publishing Co., Singapore, 2003.

[4] A. Egyed. Resolving uncertainties during trace analysis. *Proceedings of the 12th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 3–12, November 2004.

[5] A. Ghabi and A. Egyed. Exploiting traceability uncertainty between architectural models and code. *In Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference*, pages 171–180.

[6] M. Lindval and K. Sandahl. Practical implications of traceability. *Software Practice and Experience*, 26(10):1161–1180, 1996.

[7] A. Egyed, F. Graf, and P. Grünbacher. Effort and quality of recovering requirements-to-code traces: Two exploratory experiments. *In 2010 18th IEEE International Requirements Engineering Conference*, pages 221–230, September 2010.

[8] P. Mäder and A. Egyed. Assessing the effect of requirements traceability for software maintenance. *ICSM 2012, 28th IEEE International Conference on Software Maintenance*, 2012.

[9] Regan, Gilbert, Mc Caffery, Fergal, McDaid, Kevin, Flood, and Derek. Traceability-why do it? *SPICE 2012*, May 2012.

[10] A. Von Knethen. Automatic change support based on a trace model. *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'02)*, September 2002.

[11] A. Von Knethen, B. Paech, F. Kiedaisch, and F. Houdek. Systematic requirements recycling through abstraction and traceability. *Proceedings of the IEEE International Requirements Engineering Conference*, September 2002.

[12] P. Constantopoulos, M. Jarke, Y. Mylopoulos, and Y. Vassiliou. The software information base: A server for reuse. *VLDB Journal, 4(1)*, pages 1–43, 1995.

[13] A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 9(2), February 2003.

[14] F. Pinheiro and J. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, pages 52–64, March 1996.

[15] O. Gotel and P. Mäder. Towards automated traceability maintenance. *Journal of Systems and Software*, 85(10):2205–2227, 2011.

[16] J. Cleland-Huang, C. Chang, and J. Wise. Supporting event based traceability through high-level recognition of change events. *Proceedings of IEEE COMPSAC Conference*, August 2002.

[17] L.G.P Murta, A. Van der Hoek, and C.M.L Werner. Archtrace: policy-based support for managing evolving architecture-to-implementation traceability links. *In: 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06*, pages 135–144, 2006.

[18] K. Dohyung. Java mpeg player. *http://peace.snu.ac.kr/dhkim/java/MPEG/*, 1999.

[19] A. Nöhrer and A. Egyed. Conflict resolution strategies during product configuration. *In Fourth International Workshop on Variability Modelling of Software-Intensive Systems*, pages 107–114, 2010.

[20] A. Nöhrer, A. Biere, and A. Egyed. A comparison of strategies for tolerating inconsistencies during decision-making. *In Proceedings of the 16th International Software Product Line Conference - Volume 1*, pages 11–20, 2012.

[21] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reason.*, 40(1):1–33, 2008.

[22] A. Nöhrer, A. Biere, and A. Egyed. Managing sat inconsistencies with humus. *In U. W. Eisenecker, S. Apel, and S. Gnesi, editors, VaMoS*, pages 83–91, 2012.

EGOR EROFEEV

PERSONAL INFORMATION

Born in Russia, 12 January 1985

address Inzell 6, 4083 Haibach, Austria

email egor.erofeev@gmail.com

phone +43 676 930 6512

marital status single

WORK EXPERIENCE

2006–2009 Web developer, UREC

UREC

Completed projects with different levels of difficulty (simple web-sites, web-catalogs, online shopping, etc.) in a small team. Led negotiations with customers, analyzed goals, designed requirements specification, acted as a project manager for the team of 3 people. Performed adaptation of content management systems, integration with customer's information systems, development.

EDUCATION

2009–present Johannes Keppler University, Linz

Diplomingenieur

Branch of study: Software Engineering
Thesis: *Traceability decision making in the case of the presence of conflicts*(in work)
Advisors: Univ.-Prof. Dr. Alexander Egyed M. Sc.

2002-2007 Ufa State Aviation Technical University

Diplomingenieur

Branch of study: Computer Science and Information Systems
Thesis: *Irregular allocation of polygones in semi-infinite strip using methods of nonlinear optimization*

2006 TU Dresden

6 weeks summer practice

Topic: Computation of allocation areas
Supervisor: Dr. Guntram Scheithauer

TECHNICAL SKILLS AND EXPERIENCE

Operating systems Windows, basic knowledge in Linux(Debian, Ubuntu)

Programming languages Java, C, C++. Familiar with Ada, Haskell, Smalltalk, VHDL

Scripting languages JavaScript, PHP

Architecture & design principles Object-oriented programming, procedural programming, object-oriented design, design patterns, familiar with parallel computations and functional programming

Tools Eclipse, SVN, LaTeX, Visual Studio, Visio

Other SQL, SAT Solvers, XML/XSD, Semantic technologies (RDF/OWL, SPARQL), UML

OTHER INFORMATION

Languages

Russian · Mothertongue

German · Intermediate (conversationally fluent)

English · Intermediate (conversationally fluent)

October 5, 2013

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäßentnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, Oktober 2013

Egor Erofeev