Master's Thesis

On the Usage of Value- and Dependency-based Models for Spreadsheet Debugging with SMT Solvers

Andrea Höfler
andrea.hoefler@student.tugraz.at

Institute for Software Technology (IST)
Graz University of Technology
Inffeldgasse 16B/II,
8010 Graz, Austria

Supervisors: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa,
Dipl.-Ing. Dr.techn. Birgit Hofer

Graz, February 2015

Abstract

Spreadsheet programs count among the most used end-user programs. They are vital for many businesses, but also commonly used by private people. Therefore, especially because these programs are so vastly used, it would be preferable that spreadsheets are free from errors. This however, is rarely the case, showing the importance of spreadsheet debugging. There are many different approaches for spreadsheet debugging. However, they are hardly used in practice, due to unreasonable debugging time or unsatisfying results. Therefore, in this work we extend a framework developed by a team at the Graz University of Technology with a spreadsheet debugging approach based on the principles of Model-based Software Debugging (MBSD) with dependency-based models in combination with Z3, a Satisfiability Modulo Theories (SMT) solver. Additionally, we integrate a method to verify dependency-based diagnoses with value-based models to improve the diagnoses' qualities. Furthermore, we compare our dependency-based approaches with the framework's existing value-based approach. In doing so we show 1) that on average debugging spreadsheets with dependency-based models is considerably faster than with value-based models, 2) that the quality of the dependency-based diagnoses can be improved to equal that of the value-based approach when verifying the diagnoses with value-based models, 3) that in the best cases on average around ten cells need to be inspected to find a faulty cell among the reported diagnoses, and 4) that there is a correlation between a low diagnoses quality and a high debugging time. Based on these findings it is possible to further improve the runtime of the dependency-based approaches and the quality of their reported diagnoses. Furthermore, due to the extension of dependency-based models to the framework it is now possible to integrate any Boolean Satisfiability (SAT)- or SMT solver and compare its performance to Z3 when debugging spreadsheets.

Kurzfassung

Tabellenkalkulationsprogramme zählen zu den meist genutzten Enduser-Anwendungen. Sie sind aus Unternehmen nicht wegzudenken und werden auch im privaten Sektor sehr häufig verwendet. Genau aus diesem Grund, da diese Programme so weit verbreitet sind, wäre es wichtig, dass Tabellenkalkulationen frei von Fehlern sind. Leider ist das nur selten der Fall, was die Wichtigkeit von Tabellenkalkulationsdebugging aufzeigt. Es gibt viele unterschiedliche Ansätze um Tabellenkalkulationen zu debuggen, allerdings werden sie kaum in der Praxis genutzt. Das liegt unter anderem daran, dass die Debug-Zeit zu lange oder das Ergebnis zu unzufriedenstellend ist. Aus diesem Grund erweitern wir in dieser Arbeit ein Framework, entwickelt von einem Team an der Technischen Universität Graz, um eine Methode zum Debuggen von Tabellenkalkulationen. Diese Methode basiert auf den Grundlagen von Model-based Software Debugging (MBSD) mit abhängigkeits-basierten Modellen in Kombination mit Z3, einem Satisfiability Modulo Theories (SMT) Solver. Zusätzlich fügen wir dem Framework eine Methode hinzu um abhängigkeits-basierte Diagnosen mit wert-basierten Modellen zu überprüfen und somit die Qualität der Diagnosen zu steigern. Weiters vergleichen wir die abhängigkeits-basierten Ansätze mit der—bereits im Framework integrierten—wert-basierten Methode. Dabei zeigen wir, 1) dass im Durchschnitt das Debuggen von Tabellenkalkulationen mit abhängigkeits-basierten Modellen deutlich schneller ist als mit wert-basierten Modellen, 2) dass mit wert-basierter Überprüfung die Qualität der abhängigkeits-basierten Diagnosen soweit verbessert werden kann, dass sie gleich der Qualität der wert-basierten Diagnosen ist, 3) dass im besten Fall durchschnittlich ungefähr zehn Zellen untersucht werden müssen, bis eine falsche Zelle in den retournierten Diagnosen gefunden wird, und 4) dass es einen Zusammenhang zwischen einer niedrigen Diagnosen Qualität und einer hohen Debug-Zeit gibt. Basierend auf diesen Ergebnissen ist es möglich die Debug-Zeit der abhängigkeits-basierten Methoden und die Qualität der Ergebnisse zu verbessern. Zusätzlich ist es durch die Erweiterung am Framework möglich jeden Boolean Satisfiability (SAT)- oder SMT Solver ins Framework zu integrieren und seine Leistung beim Debuggen von Tabellenkalkulationen mit der von Z3 zu vergleichen.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

....

Datum Unterschrift

Contents

1 Introduction **1**

2 Basic Definitions **5**

3 Constraint-, SAT- and SMT Solvers **10**

3.1 Constraint Solver 10

3.1.1 Constraint Satisfaction Problem 11

3.1.2 Famous Problems expressed as CSP 11

3.1.3 Resolution of CSPs 13

3.2 SAT Solver 17

3.2.1 Propositional Logic 17

3.2.2 Boolean Satisfiability Problem 18

3.2.3 Famous Problems expressed as SAT 18

3.2.4 Davis-Putnam-Logemann-Loveland Paradigm 19

3.3 SMT Solver 22

3.3.1 First-order Logic 23

3.3.2 Satisfiability Modulo Theories Problem 24

3.3.3 Theories 25

3.3.4 Famous Problems expressed as SMT 33

3.3.5 Resolution of SMT 34

3.3.6 DPLL(T) Paradigm 36

3.4 Conclusion 40

4 SMT Solver Comparison **41**

4.1 Z3 42

4.2 CVC4 46

4.3 MathSAT 5 47

4.4 SMTInterpol 50

4.5 veriT . . . 51
4.6 Yices 2 . . . 52
4.7 Findings . . . 53

5 Framework and Implementation 55
5.1 Existing . . . 55
5.1.1 Model-based Software Debugging . . . 56
5.1.2 Z3's Solving Methodologies . . . 61
5.1.3 Supported Spreadsheet Functions . . . 67
5.2 Extensions . . . 69
5.2.1 Dependency-based Models . . . 69
5.2.2 Sophisticated Dependency-based Model . . . 74
5.2.3 Verifying Diagnoses with Value-based models . . . 78
5.2.4 Extended Spreadsheet Functions . . . 82

6 Empirical Evaluation 86
6.1 Spreadsheet Corpus . . . 86
6.2 Evaluation Results . . . 88
6.2.1 Runtime Comparison . . . 90
6.2.2 Diagnosis Comparison . . . 91
6.2.3 Faulty Cells' Distribution . . . 100

7 Related Work 105

8 Conclusion 108

List of Figures 111

List of Tables 112

List of Algorithms 113

Acronyms 114

Bibliography 116

Chapter 1

Introduction

Spreadsheet programs, like Microsoft's Excel, OpenOffice's Calc or Apple's Numbers, count among the most used end-user programs. They are vital for many businesses, but also commonly used by private people. Due to their vast functionality these programs can be considered as programming environments for non-professional programmers. With help of spreadsheet programs, people can easily create very complex spreadsheets, sometimes containing thousands of formulas. Since these programs are so vastly used, it would be preferable that spreadsheets are free from errors. This however, is rarely the case, as Ray Panko shows in a study conducted in 1995 to 2007 [56]. During this study about 100 spreadsheets of different US and British companies were investigated and 88 % of these spreadsheets were erroneous. Brown and Gould [14] conducted a study, where nine highly experienced spreadsheet developers had to each develop 3 different spreadsheets. Even though these persons can be considered experts in the field of spreadsheet development, 63 % of the spreadsheets contained errors. These examples show the importance of spreadsheet debugging and even though there are many different approaches, spreadsheet debugging is hardly used in practice. Strategies that try to debug spreadsheets with the help of constraint solvers are restricted through the limited support of real numbers. Furthermore, if large spreadsheets are considered, it is difficult to debug the spreadsheets within a reasonable time span. That is where Satisfiability Modulo Theories (SMT) solvers come in handy to get rid of these limitations. SMT solvers can handle real numbers and since they operate modulo a theory, they can easily be expanded to handle many different data types. To determine

how well SMT solvers perform when debugging spreadsheets, we make use of a framework [6] developed by a team at the Graz University of Technology. This framework compares different SMT- and constraint solvers based on their execution time and performance when debugging spreadsheets. Their approach for spreadsheet debugging is called Model-based Software Debugging (MBSD) with value-based models. Until now they integrated Z3, an SMT solver, and two constraint solvers, called Choco and Minion. Their research showed that Z3, in combination with the MCSes-U algorithm [48], exceeds the constraint solvers concerning modeling abilities and execution time. On average Z3 is six times faster than Choco and Minion. Whereas, the performance difference between Choco and Minion is minimal. However, their work is mainly focused on integrating the constraint solvers and it remains unclear, whether other SMT solvers would yield similar performance as Z3 when debugging spreadsheets with the MBSD approach and value-based models. Therefore, we give a general overview of constraint-, Boolean Satisfiability (SAT)- and SMT solvers. We show how they work and how they solve specific problems. Furthermore, we conduct a comparison of different state-of-the-art SMT solvers, regarding their functionality and suitability for spreadsheet debugging. For the comparison we only consider solvers that are able to operate with real numbers and since a translation of a spreadsheet into a spreadsheet debugging problem results in a non-linear arithmetic problem, it is equally important that the SMT solvers support the theory of non-linear arithmetic. Moreover, the spreadsheet debugging algorithm MCSes-U that performed best in combination with Z3 depends on the solvers' functionality to extract unsatisfiable cores. Therefore, another important requirement is the solvers' support of unsatisfiable core extraction. Surprisingly we found that not many SMT solvers support real numbers and unsatisfiable core extraction. Furthermore, Z3 is the only solver that supports non-linear arithmetic, meaning it is currently the only SMT solver suitable for MBSD of spreadsheets with value-based models. Therefore, we introduce two different dependency-based models for MBSD of spreadsheets, based on the research of Hofer et al. [43] and integrate them into the framework [6]. Dependency-based models can be expressed in Propositional Logic (PL) and therefore, any state-of-the-art SAT solver could be used for MBSD of spreadsheets with dependency-based models. However, most SMT solvers integrate a SAT solver and therefore, we can also use any

modern SMT solver. Since debugging spreadsheets with dependency-based models mostly results in less accurate diagnoses than with the value-based approach, we furthermore, introduce a method to verify dependency-based diagnoses with value-based models to improve their quality. However, using value-based models for diagnosis verification changes the spreadsheet problem into a non-linear arithmetic problem. Therefore, the verification can only be conducted with Z3. Finally, we evaluate the different approaches. Specifically, we compare the runtimes of the approaches with one another and state that on average debugging spreadsheets with dependency-based models is considerably faster than with value-based models. The fasted approach (sophisticated without value-based diagnosis verification) for example is on average 7.4 times faster than the value-based approach. Furthermore, we compare the quality of each approach's reported diagnoses and show that the quality of the dependency-based diagnoses is slightly worse than that of the value-based approach. However, it can be improved to equal the quality of the value-based approach when verifying the diagnoses with value-based models. Furthermore, we show that there exists a correlation between a low diagnoses quality and a high runtime. At last, we give an overview of the faulty cells' distribution by means of the reported diagnoses. We show that in the best cases on average around ten cells need to be inspected to find a faulty cell among the reported diagnoses. In the average case this number increases to an average of around seventeen cells. Based on these findings it is possible to further improve the runtime of the dependency-based approaches and the quality of their reported diagnoses. Furthermore, due to the extension of dependency-based models to the framework, it is now possible to integrate any SAT- or SMT solver and compare its performance to Z3 when debugging spreadsheets.
To summarize, we state the main contributions of this master's thesis which are:

- an explanation of the basic functionality of constraint-, SAT-, and SMT solvers,
- a comparison of six state-of-the-art SMT solvers concerning their functionality,
- an extension of the framework with dependency-based models for MBSD of spreadsheets with SMT solvers,

- an introduction of a value-based verifying method to improve the quality of dependency-based diagnoses and its integration into the framework,
- an extension of the framework with additional spreadsheet functions,
- an enhancement of the list of functions for which coincidental correctness might occur, and
- an evaluation of the runtime, diagnoses quality and faulty cells' distribution of the dependency-based approaches in comparison to the value-based approach.

This master's thesis is organized as follows: Chapter 2 states the basic definition of spreadsheets and the spreadsheet language. Chapter 3 introduces the principles of constraint-, SAT- and SMT solvers. We give a detailed description of their input languages, the problems they are designed to solve and how they operate to solve these problems. Chapter 4 consists of a comparison of six state-of-the-art SMT solvers that support real numbers. We give a short summary of their basic functionality as well as describe their technical features. In Chapter 5 we describe the framework's [6] existing features and design. Furthermore, we state the extensions we added to the framework, including the dependency-based models and the value-based diagnosis verification. Chapter 6 compares the different approaches concerning their runtime, quality of diagnoses and distribution of faulty cells among the reported diagnoses. Some related work is discussed in Chapter 7 and finally, the work is concluded in Chapter 8.

Chapter 2

Basic Definitions

This chapter provides an overview of the syntax and semantics of spreadsheets and the definition of the spreadsheet debugging problem as a fault localization problem. Furthermore, we describe the differences between spreadsheet debugging and conventional software debugging, as well as how these differences can be overcome so that traditional software debugging strategies can be applied to spreadsheet debugging.

Given the complexity of functions available in spreadsheet programs it would be unreasonable to give a full definition of the spreadsheet syntax. Therefore, to keep the extent of this definition manageable we restrict it to basic arithmetic operators (+, -, ·, /, <, ...) and functions which can be easily expressed in First-Order Logic (FOL), like for example the conditional (**IF**), summation (**SUM**), minimum (**MIN**) or power (**POWER**) function. Furthermore, we base our definitions upon the fact that common spreadsheet programs only support a finite number of cells. Even though programs theoretically allow loops, we additionally restrict spreadsheets to be loop-free. Therefore, from here on the term spreadsheet signifies a finite and loop-free spreadsheet.

If we consider spreadsheets taken from Excel, Numbers or Calc, we can say that they are nothing more than a matrix consisting of cells. Each cell c has a position $\mathrm{p}(c)$, a value $\mathrm{v}(c)$ and a formula $\mathrm{f}(c)$. We define these functions and the spreadsheet syntax and semantics (Definition 2.1 to Definition 2.6) like Hofer et al. in [42].

Definition 2.1. Position: The position of a cell consists of two coordinates x and y, representing the column and row number of a cell within a spreadsheet. $\mathrm{p}_x(c)$ returns the column number and $\mathrm{p}_y(c)$ the row number.

Definition 2.2. Value: The value of a cell can either be undefined $\bigcirc$, an error $\perp$, a Boolean, an integer, a real number or a string. The value itself is determined through the evaluation of the formula $\mathrm{f}(c)$.

Definition 2.3. Formula: A formula of a cell $\mathrm{f}(c)$ can either be empty ϵ or a formula written in the language $\mathcal{L}$.

Definition 2.4. Area: An area of a spreadsheet Π is a set containing multiple cells.

$$c_1 \underline{:} c_2 = \left\{ c \in \Pi \middle| \begin{array}{l} \mathrm{p}_x(c_1) \leq \mathrm{p}_x(c) \leq \mathrm{p}_x(c_2) \wedge \\ \mathrm{p}_y(c_1) \leq \mathrm{p}_y(c) \leq \mathrm{p}_y(c_2) \end{array} \right\}$$

Above definitions describe the basic structure of spreadsheets. Definitions 2.5 and 2.6 introduce the functional language $\mathcal{L}$ which is used to represent the formulas. It takes constants and values of cells together with operators and functions as input, to compute values for other cells. Not all functions of spreadsheet programs are included in the definition, since extending the language with new operators and functions is straightforward. Furthermore, no recursive functions are allowed.

Definition 2.5. Syntax of $\mathcal{L}$: A formula $\mathrm{f}(c)$ is an expression $e \in \mathcal{L}$ if it is of the following form:

- a Boolean, integer, real number or string.
- a cell name (i.e. the position of another cell)
- an operation of the form $e_1 \ o \ e_2$, with e_1 and e_2 being expressions and $o \in \{\underline{+}, \underline{-}, \underline{\cdot}, \underline{/}, \underline{\leq}, \underline{\geq}, \underline{=}, \underline{<>}\}$
- an expression of the form $\underline{(}e\underline{)}$, where e is an expression
- a function like $\underline{\mathrm{f}(}e_1\underline{,}\ldots\underline{,}e_n\underline{)}$, with $\underline{\mathrm{f}}$ denoting functions like **IF**, **SUM**, **MIN**, **POWER**, ... and $e_1, \ldots, e_n$ being expressions.

For the definition of the semantics of $\mathcal{L}$ an interpretation function $[\![\cdot]\!]$ is introduced. It maps an expression $e \in \mathcal{L}$ to a value, which is either $\bigcirc$ if no

value can be determined, $\perp$ if a type error occurs, or a number, Boolean or string.

Definition 2.6. Semantics of $\mathcal{L}$:

- If e is a constant k, then the constant is given back as a result ($[\![e]\!] = k$).
- If e denotes a cell name c, then its value is returned ($[\![e]\!] =$v(c)).
- If e is of the form $\underline{(}e_1\underline{)}$, then $[\![e]\!] = [\![e_1]\!]$.
- If e is of the form $e_1\ o\ e_2$, then:
  - If either $[\![e_1]\!] = \perp$ or $[\![e_2]\!] = \perp$, then $[\![e_1\ o\ e_2]\!] = \perp$,
  - else if either $[\![e_1]\!] = \bigcirc$ or $[\![e_2]\!] = \bigcirc$, then $[\![e_1\ o\ e_2]\!] = \bigcirc$,
  - else if $o \in \{\underline{+}, \underline{-}, \underline{:}, \underline{/}, \underline{\leq}, \underline{\geq}, \underline{=}, \underline{<>}\}$, then

    $$[\![e_1\ o\ e_2]\!] = \begin{cases} [\![e_1]\!]\ o\ [\![e_2]\!] & \text{if all expressions evaluate to a number.} \end{cases}$$

- If e is of the form $\underline{\text{f}(}e_1\underline{,}\ldots\underline{,}e_n\underline{)}$, then the return value of function $\underline{\text{f}}$'s implementation is returned. Let $\underline{\text{f}_I}$ be the implementation of $\underline{\text{f}}$, then $[\![\ \underline{\text{f}(}e_1\underline{,}\ldots\underline{,}e_n\underline{)}\ ]\!] = \underline{\text{f}_I(}[\![e_1]\!]\underline{,}\ldots\underline{,}[\![e_n]\!]\underline{)}$. The return value of $\underline{\text{f}_I}$ might be $\perp$ if type errors or mismatches of arguments occur.

Above definitions are really straight forward since they are very similar to the semantics used in most modern spreadsheet programs. However, for the definition of the spreadsheet debugging problem we still need to define a failing test case. For that purpose we introduce Definitions 2.7 to 2.13 which are based on Hofer et al. [43].

Definition 2.7. Referenced cell: A cell c is a referenced cell, or referenced by an expression $e \in \mathcal{L}$, iff c is used in e.

Definition 2.8. Function ρ: Let $\rho(e)$ be a function, that returns a set consisting of all referenced cells for a specific expression, then $\rho(e)$, $e \in \mathcal{L}$ is defined as follows:

- If e is a constant, then $\rho(e) = \emptyset$.
- If e is a cell c, then $\rho(e) = \{c\}$.

- If $e = \underline{(}e_1\underline{)}$, then $\rho(e) = \rho(e_1)$.
- If $e = e_1\ o\ e_2$, then $\rho(e) = \rho(e_1) \cup \rho(e_2)$.
- If $e = \underline{\mathrm{f}(}e_1\underline{,} \ldots \underline{,}e_n\underline{)}$, then $\rho(e) = \bigcup_{i=1}^{n} \rho(e_i)$.

Definition 2.9. Direct data dependency: A cell c is direct data dependent on another cell c' iff c' is referenced in $\mathrm{f}(c)$.

$$dd(c', c) \Leftrightarrow c' \in \rho(\mathrm{f}(c))$$

Definition 2.10. Input cell: A cell c is an input cell, iff it is not direct data dependent on another cell c' and there exists at least one cell c'' that is direct data dependent on c. (Cells containing strings are ignored.)

$$Input(\Pi) = \{\text{cells } c : (\nexists c' : dd(c', c) \wedge \exists c'' : dd(c, c''))\}$$

Definition 2.11. Output cell: A cell c is an output cell, iff there exists no cell c' that is direct data dependent on c and c is direct data dependent on at least one other cell c''.

$$Output(\Pi) = \{\text{cells } c : (\nexists c' : dd(c, c') \wedge \exists c'' : dd(c'', c))\}$$

Definition 2.12. Test case: A test case T for a spreadsheet Π is composed of two parts: input I and output O. I is a set of tuples (c, v), with c being an input cell of Π and v being its value. O is a set of tuples (c, v_{exp}), with c being a cell of Π and v_{exp} being its expected value.

Definition 2.13. Failing test case: A test case T is a failing test case, if there exists at least one cell c where the calculated value $\mathrm{v}(c)$ differs from the expected value v_{exp}. If T is a failing test case, then O can be split into two sets, O_{wrong} and $O_{correct}$. O_{wrong} contains all output cells for which hold that $\mathrm{v}(c)$ differs from v_{exp}. $O_{correct}$ contains all output cells where the calculated value equals v_{exp} or $O_{correct} = O \setminus O_{wrong}$.

Now with above definitions it is easy to state the spreadsheet debugging problem.

Definition 2.14. Spreadsheet Debugging Problem Given a spreadsheet Π and a failing test case T related to Π, the Spreadsheet Debugging Problem (SDP) is the problem of finding the locations in Π, that could cause T to fail.

This definition states the spreadsheet debugging problem as a *fault localization problem*. This implies, that the debugging process only locates possible causes of faults, while no solution is offered. However it is possible, yet not in the scope of this work, to alternatively define the debugging problem as a fault correction problem.

Like Hofer et al. in [42] describe, there are many issues to overcome when we apply conventional software debugging techniques for traditional procedural and object-oriented programming languages to spreadsheets. For conventional programming languages, there exists the concept of code coverage to measure the lines of code executed by test cases. This concept cannot be applied to spreadsheets, since they do not have explicit lines of code and do not support test execution. To resolve these problems some modifications need to be made. The lines of code need to be mapped to the spreadsheet's cells, and so-called *cones* need to be computed as an alternative to the missing concept of code coverage in spreadsheets.

Definition 2.15. Function Cone: The Function Cone for a cell $c \in \Pi$ computes the data dependencies of c like follows:

$$\textsc{Cone}(c) = c \cup \bigcup_{c' \in \rho(c)} \textsc{Cone}(c')$$

Finally, the correctness of the output cells are either checked manually, by comparing the results of the current spreadsheet with another one which is considered correct, or automatically with a technique that detects spreadsheet "smells" [26].

Chapter 3

Constraint-, SAT- and SMT Solvers

Constraint solvers, Boolean Satisfiability (SAT) solvers and Satisfiability Modulo Theories (SMT) solvers are terms that often occur together in scientific papers. The technologies behind these terms are based on the same principles, also the fields of application for these solvers are closely related. Therefore, for people not familiar with topics concerning these technologies, it is difficult to understand the differences, unless it is understood how they work and what they are designed to do. Therefore, in this chapter we give a short overview of the different solvers, how they work and what kind of problems they are designed to solve.

3.1 Constraint Solver

In the last few years Constraint Programming (CP) became a more and more important topic not only in the field of software development, but in general. The broad application area of CP let many different companies to exploit this technology to develop a vast number of industrial applications for distribution planning, production planning and scheduling, tour planning, personnel allocation and many more [63, p. 11-16].
Constraint solvers take a constraint formula as input and try to find an assignment for every variable which makes the formula satisfiable. Problems like that are called constraint satisfaction problems.

3.1.1 Constraint Satisfaction Problem

Constraint Satisfaction Problems (CSPs) deal, as the name suggests, with constraints. Examples for such constraints could be temporal constraints, constraints by law, or a maximum capacity constraint. To solve a CSP all the imposed constraints have to be considered.

Definition 3.1. As the Cork Constraint Computation Centre [18] and Brailsford et al. [13] described, a CSP consists of a set of variables $X_1, X_2, \ldots, X_n$ and a set of domains $D_1, D_2, \ldots, D_n$. Each domain D_i corresponds to a variable X_i and contains a range of valid values for the corresponding variable. Furthermore, CSPs consist of conditions called constraints, which define the relations of the variables and therefore, restrict the values that each variable can simultaneously take.

A solution to a CSP is an assignment to every variable with a value from its domain. If at least one solution is found the problem is satisfiable. On the contrary, if no assignment can be found that satisfies all constraints, then the problem is unsatisfiable. Sometimes finding only one solution is not sufficient. Therefore, all possible solutions need to be found, as is the case for example if constraint solver are utilized for debugging purposes.

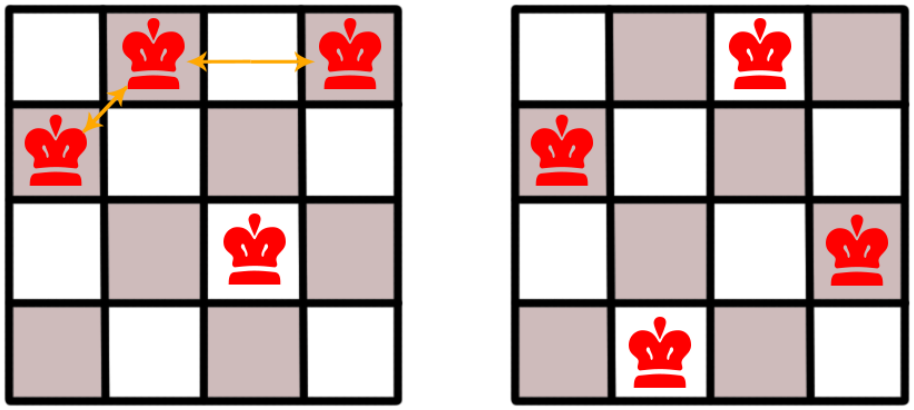
3.1.2 Famous Problems expressed as CSP

Many logic puzzles can be expressed as CSPs, like Sudoku, Numbrix, the N-queens puzzle and the map coloring problem. As a short demonstration we describe the later two problems and have a look at how they can be expressed as CSPs.

N-Queens Puzzle

The N-queens puzzle is based on the rules of chess. Given an $n \cdot n$ chess board, n chess queens have to be placed on that board in such a way, that no queen can attack another. (Queens can attack other pieces if they are in the same row, column, or diagonal from the queen.)
A 4-queens puzzle represented as a CSP can look as follows [18]:

- **Variables:** Q_0, Q_1, Q_2, Q_3 (each queen represents a row; Q_0 = queen in row 0)



(a) A wrong solution for the 4-queens puzzle.

(b) A valid solution for the 4-queens puzzle.

Figure 3.1: Example of a 4-queens puzzle.

- **Domains:** $D_i = \{0, 1, 2, 3\}$ (columns)
- **Constraints:** no queen can attack another queen

  $\textsc{Alldifferent}(Q_0, Q_1, Q_2, Q_3) \wedge$ for $i = 0 \ldots 3 \wedge j = (i+1) \ldots 3$, $k = j - i$, $Q_i \neq Q_j + k \wedge Q_i \neq Q_j - k$

Figure 3.1 shows an example for a wrong solution and a possible valid solution of the 4-queens puzzle.

Map Coloring Problem

Given a map of a country with different territories and n different colors, color the map in a way that no neighboring territories have the same color. Expressed as a CSP for a map of Austria and with three different colors this might look as follows [44]:

- **Variables:** W, N, O, ST, B, S, K, T, V (one variable for each territory)
- **Domains:** $D_i = \{$*red, green, blue*$\}$
- **Constraints:** neighboring territories must have different colors

  (V, T) $\in \{$(*red, green*), (*red, blue*), (*green, red*), (*green, blue*), $\ldots\}$ $\wedge$ (T, S) $\in \{$(*red, green*), (*red, blue*), (*green, red*), (*green, blue*), $\ldots\}$ $\wedge \ldots$ (for each combination of neighboring territories)

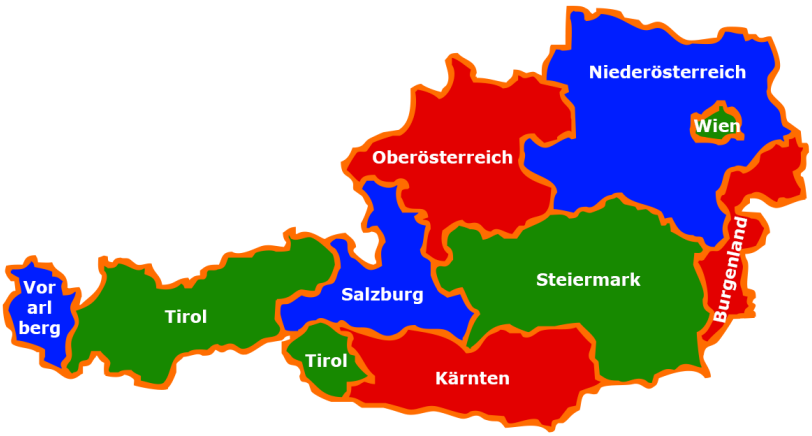


Figure 3.2: A possible solution of the map coloring problem for the map of Austria.

Figure 3.2 shows a possible solution of the map coloring problem for the map of Austria.

3.1.3 Resolution of CSPs

This section describes in a general way how constraint solvers solve CSPs. A more elaborate explanation can be found under [18] or [67, ch. 1-2]. Generally, solving a CSP means to find an assignment for all variables without violating any of the corresponding constraints. Possible values for each variable are chosen from the variable's domain. Domains differ depending on the type of the variable. For instance, a domain could include only Boolean values, integer numbers, real numbers, a mix of these or others. The solving progress itself typically involves a form of search and other various techniques like backtracking, constraint propagation and local search.

Search

The search starts with choosing a variable and assigns a possible value from its domain to it. This is continued until an assignment of all variables is found that fulfills the given formula of constraints. In fact, this is the best case scenario that hardly ever occurs. It is much more likely that at some point it is no longer possible to assign a valid value to the remaining variables due to a conflict with the constraints. This does not necessarily mean the formula is unsatisfiable. Although, this could be the case, it is much more

likely that one or more variables got assigned a wrong value. During this situation constraint solvers use a certain technique called backtracking.

Backtracking

Let us consider a simple depth-first search like described above. At some point during the search it may no longer be possible to assign a value to a variable due to the imposed constraints. In that case we must go backward. We know that a domain holds all possible values which can be assigned to a variable. When during the search a variable is assigned a value, this value is chosen randomly from the variable's domain. If furthermore, the assignment is consistent with the constraints imposed on the variable, another variable is chosen and the assignment process is repeated. In case that the assignment is not consistent, another randomly selected value from the domain gets chosen. If all values of the domain lead to an inconsistent assignment backtracking occurs. This means, the algorithm goes back a level in the search tree and assigned variables from a lower level get their values deleted. The previously assigned variable gets assigned different values from its domain and the search algorithm continues to assign values to uninitialized variables. If again no valid assignment for the input formula can be found, backtracking goes back yet another level and tries different values for that variable. This is repeated until a solution is found, or until the algorithm cannot go back further. In that case the formula is unsatisfiable. Figure 3.3 illustrates how backtracking works for a simple version of the n-queens problem.

Constraint propagation

Above described techniques are able to find solutions for CSPs, yet the efficiency of these procedures can be further improved with the help of constraint propagation techniques. One specific form of constraint propagation is called forward checking. Every time a variable gets assigned a value, the algorithm retrieves all other variables that are connected by constraints to the currently instantiated variable and adjusts their domains. Meaning values that are inconsistent with the current assignment are temporarily removed from the domains. If in any case a domain becomes empty, then another value must be chosen for the current assignment, since it is no longer

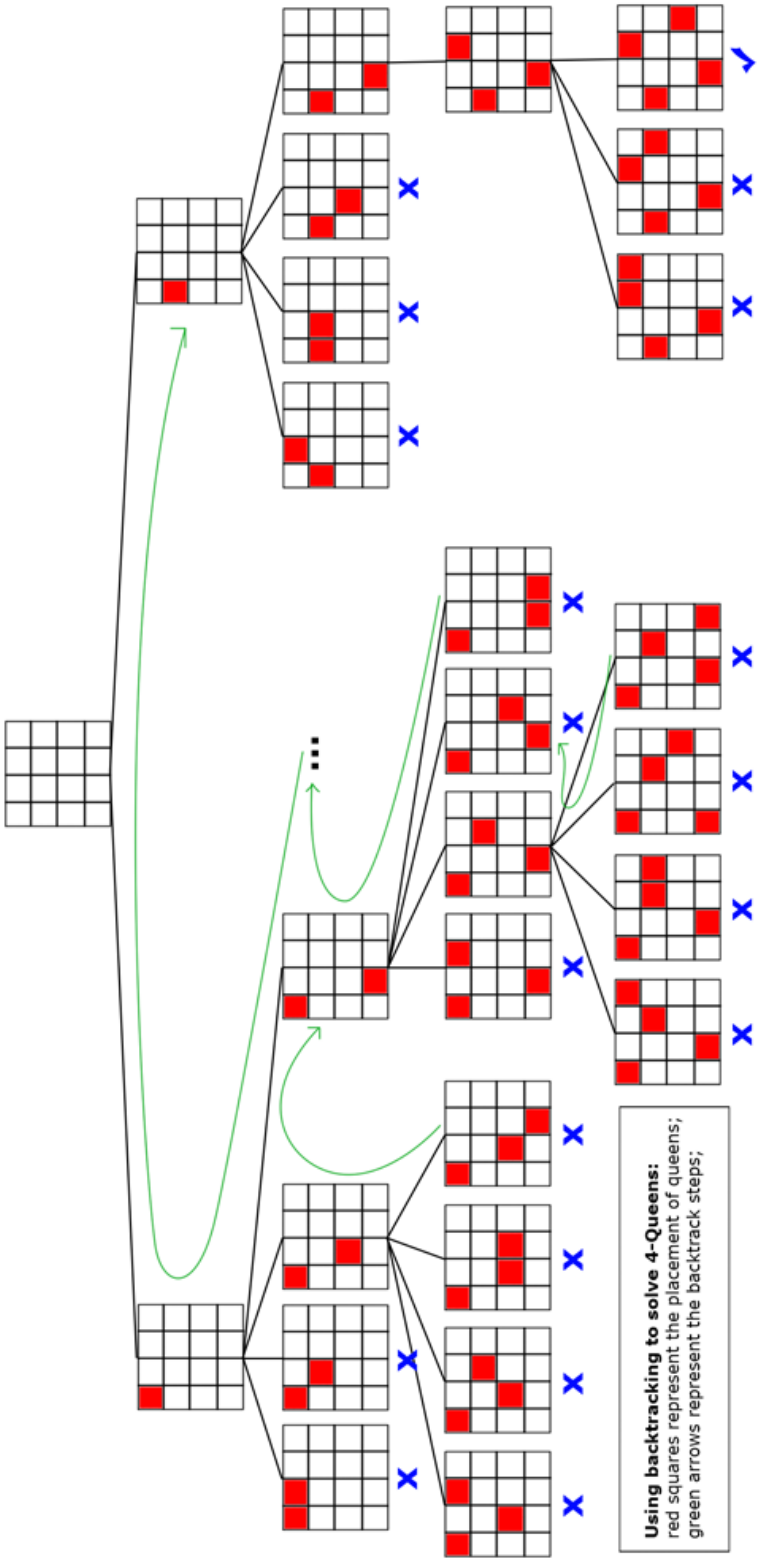


Figure 3.3: Illustration of backtracking with 4-Queens.

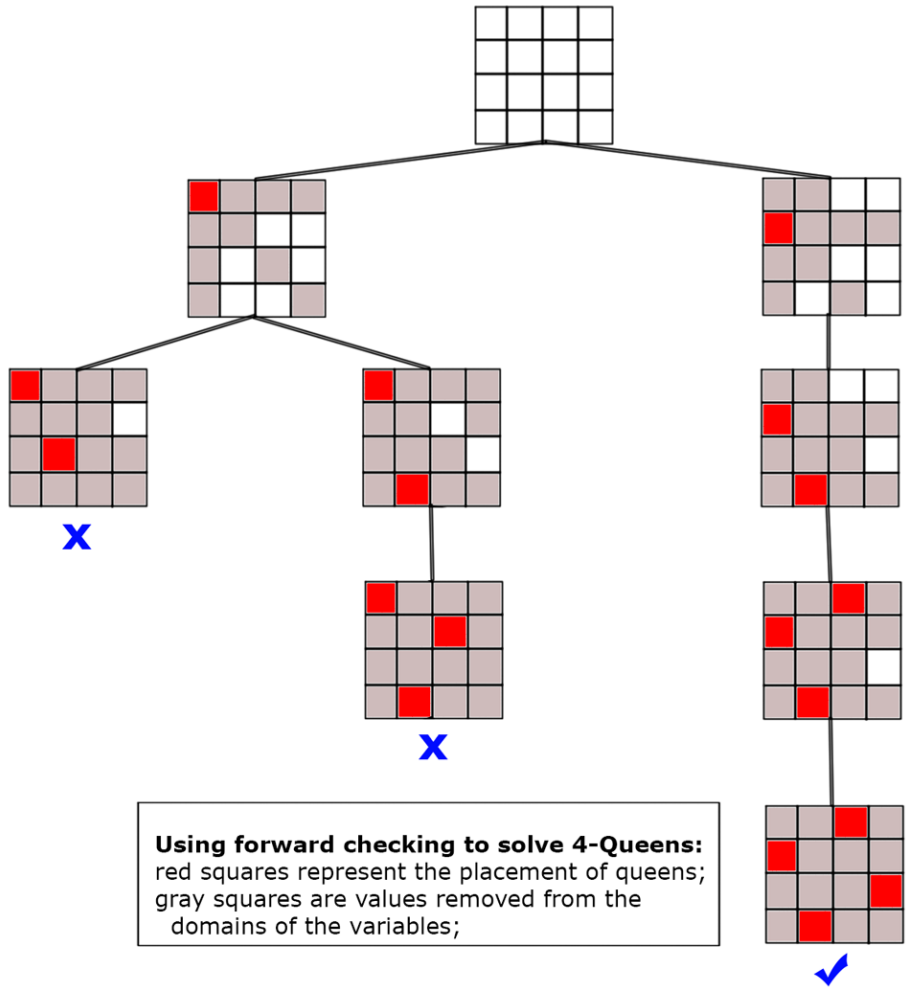


Figure 3.4: Illustration of forward checking with 4-Queens.

possible to assign all variables a value. In the event that no other value can be chosen for the current variable, the algorithm has to backtrack and the values temporarily removed from the different domains have to be restored. That is how forward checking can—to a limited extent—predict which assignments will lead to failures and act accordingly to increase efficiency. Figure 3.4 shows how the above algorithm of forward checking works, again based on the n-queens problem.

Local search

Local search first assigns a random value to all variables. This will most likely lead to an inconsistent state. Therefore, it continues with switching the values of the variables until a solution is found. It is important to mention, that local search does not work well with all types of CSPs, but can find a solution very quickly for some problems. For local search to work well a good initial assignment is critical. In fact local search works best, if the initial assignment almost solves the problem.

3.2 SAT Solver

Boolean Satisfiability (SAT) solvers specialize in solving SAT problems. SAT is a very important topic in computer science. It was the first decision problem proved to be Nondeterministic Polynomial (NP) complete (1971 by Stephen Cook and Leonid Levin [70]). In short this means there is not yet an algorithm known that efficiently solves all instances of SAT. Of course, a theory like that would lead to a development of many different solvers, each trying to disprove it. However, with no success to this day. Yet over the last decade many efficient algorithms were developed, leading to us being able to solve instances involving tens of thousands of variables and millions of constraints [62].

3.2.1 Propositional Logic

SAT solvers operate on the language of PL, which Alessandro Farinelli defined in his lecture notes on propositional and first-order logic [35]:

Definition 3.2. Variables: The language of PL consists of Boolean variables.

Definition 3.3. Operators: In addition to variables, PL makes use of the operators negation ($\neg$), conjunction ($\wedge$) and disjunction ($\vee$).

Definition 3.4. Parentheses: Essentially each finite possible sentence constructed by operators must be enclosed in parentheses. Many parentheses can be omitted though, due to operator priorities and thus improve readability. Priorities from highest to lowest are: $\neg$, $\wedge$, $\vee$.

Definition 3.5. Literals: A literal is a variable or its negation.

Definition 3.6. Terms: A term is a string of literals connected by conjunctions.

Definition 3.7. Atoms or formulas: an atom is a formula.

- If P is a formula, $\neg P$ is a formula.
- If P_1 and P_2 are formulas, $P_1 \wedge P_2$ is a formula.
- If P_1 and P_2 are formulas, $P_1 \vee P_2$ is a formula.
- Every finite concatenation of above rules is formula as well.

3.2.2 Boolean Satisfiability Problem

The Boolean or propositional satisfiability problem (SAT) describes the problem of determining whether the variables of a given Boolean formula can be assigned in such a way that the formula evaluates to TRUE. In other words, it decides if a Boolean formula is satisfiable (at least one instantiation evaluates to true) or unsatisfiable (all instantiations evaluate to false). The function itself and all its variables are all binary valued. In some special cases, especially, in computational complexity theory, SAT is restricted to be in the Conjunctive Normal Form (CNF). This means an AND ($\wedge$) of ORs ($\vee$), whereas each OR term is called a **clause**.

Example 3.1. Examples of CNFs:

- $(A \vee B) \wedge (C \vee A)$
- $(A \vee B \vee \neg C) \wedge A \wedge D$
- $\neg A \wedge B \wedge (C \vee D)$

3.2.3 Famous Problems expressed as SAT

Like for CSPs we show how the N-queens puzzle and the map coloring problem can look like, when expressed as a SAT problem.

N-Queens Puzzle

One possible option of the 4-queens puzzle expressed as a SAT problem follows:

- **Variables:** $p_{11}, p_{12}, p_{13}, p_{14}, p_{21}, \ldots, p_{43}, p_{44}$ (one variable for each field of the board)
- **Domains:** D_i={0, 1} (since the variables are of type Boolean)
- **Constraints:** no queen can attack another queen

  $((p_{11} \vee p_{12} \vee p_{13} \vee p_{14}) \wedge (\neg(p_{11} \wedge p_{12}) \wedge \neg(p_{11} \wedge p_{13}) \wedge \neg(p_{11} \wedge p_{14}) \wedge \neg(p_{12} \wedge p_{13}) \wedge \neg(p_{12} \wedge p_{14}) \wedge \neg(p_{13} \wedge p_{14}))) \wedge \ldots$ (for each row and column) $\wedge$ $(\neg(p_{11} \wedge p_{22}) \wedge \neg(p_{11} \wedge p_{33}) \wedge \neg(p_{11} \wedge p_{44}) \wedge \neg(p_{22} \wedge p_{33}) \wedge \neg(p_{22} \wedge p_{44}) \wedge \neg(p_{33} \wedge p_{44})) \wedge \ldots$ (for each diagonal)

Map Coloring Problem

For the map coloring problem we again use a map of Austria with three different colors to show one possible way of how the problem can be expressed as a SAT instance.

- **Variables:** Wr, Wb, Wg, Nr, Nb, Ng, Or, Ob, Og, STr, STb, STg, Br, Bb, Bg, Sr, Sb, Sg, Kr, Kb, Kg, Tr, Tb, Tg, Vr, Vb, Vg (one variable for each territory and color)
- **Domains:** $D_i = \{0, 1\}$ (since the variables are of type Boolean)
- **Constraints:** neighboring territories must have different colors

  $((\text{Wr} \vee \text{Wg} \vee \text{Wb}) \wedge \neg(\text{Wr} \wedge \text{Wg} \wedge \text{Wb}) \wedge \neg(\text{Wr} \wedge \text{Wg}) \wedge \neg(\text{Wr} \wedge \text{Wb}) \wedge \neg(\text{Wg} \wedge \text{Wb})) \wedge \ldots$ (for all colors per territory)) $\wedge$ $((\text{Wr} \vee \text{Nr} \vee \text{Or} \vee \text{STr} \vee \text{Br} \vee \text{Sr} \vee \text{Kr} \vee \text{Tr} \vee \text{Vr}) \wedge \ldots$ (for each color)) $\wedge$ $(\neg(\text{Vr} \wedge \text{Tr}) \wedge \neg(\text{Tr} \wedge \text{Sr}) \wedge \neg(\text{Tr} \wedge \text{Kr}) \wedge \neg(\text{Sr} \wedge \text{Or}) \wedge \neg(\text{Sr} \wedge \text{STr}) \wedge \neg(\text{Sr} \wedge \text{Kr}) \wedge \neg(\text{Or} \wedge \text{Nr}) \wedge \neg(\text{Or} \wedge \text{STr}) \wedge \neg(\text{STr} \wedge \text{Nr}) \wedge \neg(\text{STr} \wedge \text{Br}) \wedge \neg(\text{STr} \wedge \text{Kr}) \wedge \neg(\text{Nr} \wedge \text{Wr}) \wedge \neg(\text{Nr} \wedge \text{Br}) \wedge \ldots$ (for each color))

3.2.4 Davis-Putnam-Logemann-Loveland Paradigm

The Davis-Putnam-Logemann-Loveland (DPLL) procedure was introduced in 1962 by Martin Davis, Hilary Putnam, George Logemann and Donald Loveland to decide satisfiability of PL formulas in CNF, later known as SAT. Nowadays, over 50 years later, different variations of the DPLL procedure build the basis for most state-of-the-art SAT solvers [28], [27]. It consists of the following transition rules, which describe in a general way how DPLL-based SAT solvers work.

UnitPropagate

$$M \parallel F, C \vee l \implies M\ l \parallel F, C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \text{ is undefined in } M \end{cases}$$

For a CNF formula to be satisfiable, all its clauses have to be true. Therefore, **UnitPropagate** looks for clauses whose literals have all been assigned the value false, with exception of one literal, whose value is not yet defined in model M. The only way for the clause to be true in M is to extend M with the remaining literal equal to true [54].

PureLiteral

$$M \parallel F \implies M\ l \parallel F \quad \textbf{if} \begin{cases} l \text{ occurs in some clause of } F \\ \neg l \text{ occurs in no clause of } F \\ l \text{ is undefined in } M \end{cases}$$

A literal l is called pure, if its negation does not occur in the formula F. Leading to the consequence, that F is only satisfiable if l is true. Therefore, M has to be extended to make l true [54].

Decide

$$M \parallel F \implies M\ l^d \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$$

Decide conducts a case split. It chooses an undefined literal l, assigns a truth value to it and adds it to M. Additionally, l gets denoted as a *decision literal* l^d. In case that $l \in M$ cannot be extended to a model of F, $\neg l \in M$ must still be considered [54].

Fail

$$M \parallel F, C \implies \mathit{FailState} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \text{ contains no decision literals} \end{cases}$$

If a *conflicting clause* gets detected and M contains no decision literals, DPLL produces a *FailState*. Meaning it returns the result that F is unsatisfiable [54].

Backtrack

$$M\ l^d\ N \parallel F,\ C \implies M\ \neg l \parallel F,\ C \quad \textbf{if} \begin{cases} M\ l^d\ N \models \neg C \\ N \text{ contains no decision literals} \end{cases}$$

If a *conflicting clause* is detected but no *FailState* is produced, the **Backtrack** procedure goes back one *decision level*, by changing the most recent decision literal l^d to $\neg l$ and removing all literals from M that got added after l^d. Furthermore, $\neg l$ is added to M as a non-decision literal, since the other possibility has already been tested [54].

Above procedure describes only the theoretical principles of the classical DPLL algorithm. Modern implementations do not implement this version of the DPLL algorithm, but an improved one, where the **PureLiteral** rule is mostly used in the preprocessing step and **Backjumping** is used instead of chronological **Backtracking**. Furthermore, most DPLL implementations use the backjump clauses by adding them to the clause set as learned clauses (*lemmas*). This process is usually called Conflict-Driven Clause Learning (CDCL). Modern implementations also make use of a **Restart** rule, which restarts the DPLL procedure when the search is not making enough progress. So for a modern DPLL implementation we have the rules **UnitPropagate, Decide** and **Fail** from the classic DPLL, and in addition the rules **Backjump, Learn, Forget** and **Restart** [54].

Backjump

$$M\ l^d\ N \parallel F,\ C \implies M\ l' \parallel F,\ C \quad \textbf{if} \begin{cases} M\ l^d\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F,\ C \models C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^d\ N \end{cases}$$

Chronological backtracking always goes back to the last decision literal l^d and negates it to $\neg l$. Conflict-driven **Backjumping** evaluates why the conflicting clause was produced and then, if necessary, goes back *several* decision levels at once, where it adds some new literals to that lower level. This process is much more efficient than the backtracking method, since it jumps over levels unrelated to the conflict [54].

Learn

$$M \parallel F \implies M \parallel F,\ C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models C \end{cases}$$

Learn basically uses backjump clauses $(C' \vee l')$ and adds them to the clause set as learned clauses. Theoretically, **Learn** allows to add any clause C to F, as long as all atoms of C are included in either F or M. Meaning, not only lemmas can be added, but any produced consequence of F [54].

Forget

$$M \parallel F,\ C \implies M \parallel F \quad \textbf{if} \begin{cases} F \models C \end{cases}$$

Forget, on the contrary to **Learn**, removes lemmas with relevance or activity levels below a certain threshold. For instance, if a clause has not become a unit or conflicting clause for n iterations, it gets removed. Similar to **Learn**, **Forget** theoretically can remove not just those clauses added by **Learn**, but any clause, if it is entailed by the rest of F [54].

Since producing consequences and determining entailments are very costly, their usage is limited in practice.

Restart

$$M \parallel F \implies \emptyset \parallel F$$

The idea behind **Restart** is that the additional knowledge of the learned lemmas will lead to the **Decide** rule to behave differently and find a solution faster than by backtracking [54].

3.3 SMT Solver

As the name Satisfiability Modulo Theories (SMT) suggests, SMT solvers have a close relation to Boolean Satisfiability (SAT). In fact, most SMT solvers use a state-of-the-art SAT solver to evaluate whether an SMT instance is satisfiable or not. That is why recent breakthroughs in SAT solver development also resulted in a great advancement in the relevance of SMT solvers, leading to the development of many different industrial applications

in the fields of software verification, model-based testing, model checking, test-case generation and many more [53], [70].
Within this section we describe the basic technology of SMT solvers and the kind of problems they are designed to solve.

3.3.1 First-order Logic

Generally speaking, SMT solvers determine whether a formula, in the language of quantifier-free First-Order Logic (FOL), is satisfiable or not. Only a few SMT solvers are able to handle quantifiers. That is why both quantifiers, for all ($\forall$) and there exists ($\exists$), were omitted in the below definition of FOL. For a complete definition of FOL we refer to Alessandro Farinelli's lecture notes on propositional and first-order logic [35]. Definitions 3.8 to 3.16, are based on Farinelli's description of FOL [35].

Definition 3.8. Variables and Constants: The language of FOL consists of constants and variables with values of various types, f.i. Boolean, integer or real.

Definition 3.9. Operators: Additionally, to the operators of PL, negation ($\neg$), conjunction ($\wedge$) and disjunction ($\vee$), FOL makes use of the equals operator ($=$).

Definition 3.10. Parentheses: As already stated in Definition 3.4 for the language of PL, parentheses can be omitted due to operator priorities. The same is true for FOL with priorities from highest to lowest: $=$, $\neg$, $\wedge$, $\vee$.

Definition 3.11. Predicates: Predicate symbols, also called relation symbols, are most of the time denoted by uppercase letters. They have an arity stating how many parameters a predicate takes. Basically, a predicate is a statement that is either *true* or *false*, depending on the values of its arguments. Predicates of arity 0 are equivalent with Boolean variables.

Example 3.2. Assuming, *Person* is a predicate symbol with arity 1, then $Person(x)$ would evaluate to true only if x really is a person.

Example 3.3. Assuming, $On(Table,\ Pen)$ is a predicate with arity 2, then in case the pen is on the table it follows that $On(Table,\ Pen)$ would be true and false otherwise.

Definition 3.12. Functions: On the contrary to predicate symbols, functional symbols are mostly denoted by lowercase letters and also have an arity. Functions of arity 0 are equal to constants.

Example 3.4. Assuming, *add* is a functional symbol, then $add(x, y)$ may be interpreted as: the sum of x and y. Meaning, $add(x, y)$ returns the solution of $x + y$ as a value.

Definition 3.13. Terms: Every variable and constant on its own is a term. Furthermore, if f is a function with arity n and $t_1,\ldots,t_n$ are terms, then $f(t_1,\ldots,t_n)$ is a term as well.

Definition 3.14. Formulas: Every Boolean variable is a formula. Furthermore, if P is a predicate with arity n and $t_1,\ldots,t_n$ are terms, then $P(t_1,\ldots,t_n)$ is a formula as well. Terms linked by any operator are also formulas.

Definition 3.15. Sentences: A formula with no free variable is called a sentence.

Definition 3.16. Atomic formulas: A formula containing no logical connective and no bound variable is called an atomic formula or an atom.

3.3.2 Satisfiability Modulo Theories Problem

An SMT problem describes the problem of determining whether a formula expressed in quantifier-free FOL is satisfiable, with respect to a background theory. Typical examples of such theories are for instance: the theory of uninterpreted functions with equality, the theory of linear arithmetic over integers or reals, or the theories of different data structures like lists, arrays, bit-vectors.

Solving SMT problems draws on symbolic logic's biggest problems of the past century, namely the decision problem, complexity theory and completeness and incompleteness of logical theories. As already mentioned, SMT solvers rely on SAT solvers and SAT is NP-complete. Furthermore, FOL is undecidable and the computational complexity of most SMT problems is very high. That is why most solvers focus on solving practical problems, like formulas produced by verification and analysis tools, since these formulas can be efficiently solved. SMT additionally struggles with finding

algorithms that not only are able to efficiently handle different theories, but can also be modularly combined with one another. However, even with all these problems and limitations there has been a vast progress in the field of SMT in recent years. Many problems can be solved, not only thanks to modern SAT solvers, but also because of constantly improved algorithms and efficient implementations [52].

3.3.3 Theories

One core part of SMT is made up by the theories, or more specifically, the theory solvers. Most modern SMT solvers follow the Davis-Putnam-Logemann-Loveland modulo Theories (DPLL(T)) paradigm, which suggests a separate implementation for each theory. This leads to the conclusion that not all SMT solvers implement the same theories, just those needed for their field of application. Some theories became very popular because of their wide range of application, like the theory of uninterpreted functions with equality, linear arithmetic over integers and reals or the theories of arrays or bit-vectors. Due to their popularity, the website Satisfiability Modulo Theories Library (SMT-LIB) [9] started to provide standard rigorous descriptions of these most commonly used theories. On the website these descriptions of theories are named SMT-LIB logics. There is also a yearly competition called Satisfiability Modulo Theories Competition (SMT-COMP), where different SMT solvers compete against each other. Participants of this competition can enroll their SMT solvers in the divisions of their choice, since not every solver supports each background theory. All these different divisions correlate to a specific SMT-LIB logic. Figure 3.5 shows an overview of the SMT-LIB logics. The abbreviations' meanings of these logics are explained in Table 3.1.

Basic Theory Definitions

Basically, a theory is a set of sentences and we say, a formula φ is satisfiable *modulo* a theory T if $T \cup \{\varphi\}$ is satisfiable. Meaning, there exists a model M that satisfies φ under the theory T, denoted as $M \models_T \varphi$. Furthermore, if there is a procedure δ that checks whether any quantifier-free formula is satisfiable or not, then the satisfiability problem for a theory T is *decidable*. Meaning, δ is a *decision procedure* for T [52].

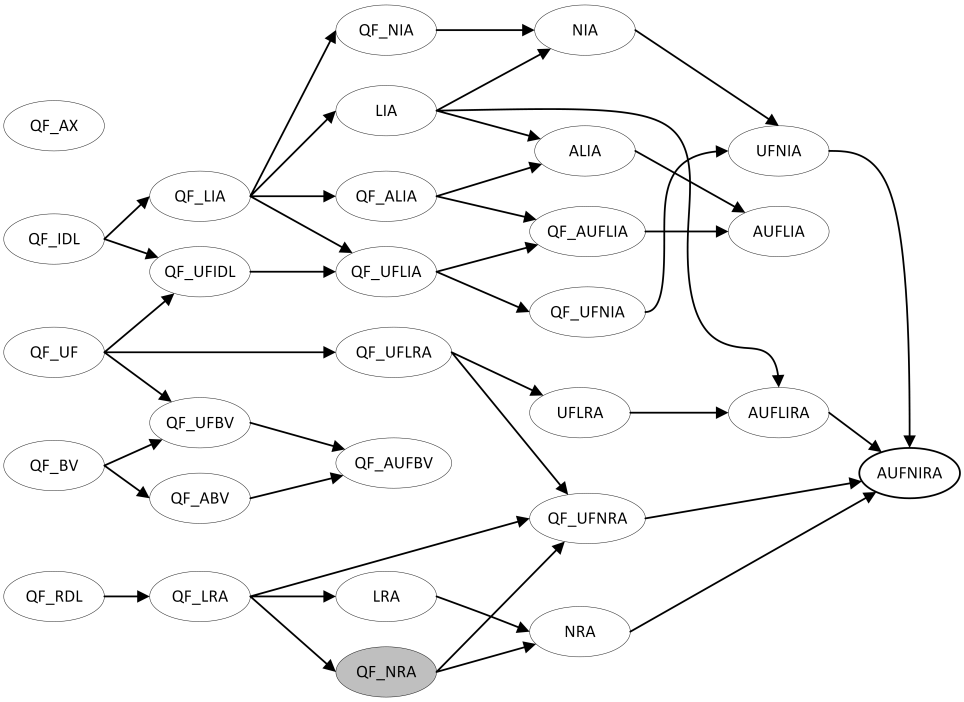


Figure 3.5: Overview of the SMT-LIB logics [9]. A link from a logic L1 to a logic L2 means that every formula of L1 is also a formula of L2. The logic shaded in gray is the one relevant for spreadsheet debugging.

Abbreviation	**Meaning**
QF	quantifier-free
A or AX	theory of arrays
BV	theory of fixed size bit-vectors
IA	theory of integer arithmetic
RA	theory of real arithmetic
IRA	theory of mixed integer real arithmetic
IDL	theory of integer difference logic
RDL	theory of real difference logic
L before IA, RA, IRA	linear
N before IA, RA, IRA	non-linear
UF	uninterpreted functions with equality

Table 3.1: Short explanation of the SMT-LIB logics' abbreviations [9].

Uninterpreted functions with equality

The theory of uninterpreted functions with equality is denoted by the SMT-LIB as QF_UF: quantifier-free uninterpreted functions with equality. An uninterpreted function is a function with a name and arity but, as the name suggests, no interpretation, like for example:

$$f(x),\ g(x,y),\ f(f(x)),\ \text{or}\ f(g(f(y),x))$$

Furthermore, as Condit and Harren stated in their lecture notes [25], the theory allows boolean connectives ($\wedge$, $\vee$, ...), equalities ($=$) and unequalities ($\neq$). Following axiom definitions are valid for $=$, as well as $\neq$ and were defined in [25]:

Definition 3.17. Reflexivity: $\frac{}{E=E}$

Definition 3.18. Transitivity: $\frac{E_1=E_2 \quad E_2=E_3}{E_1=E_3}$

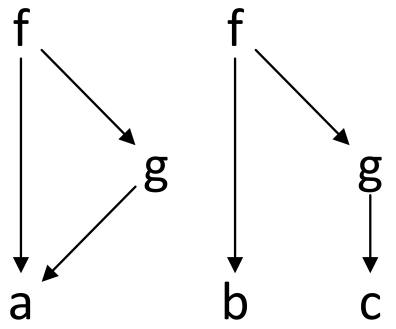
Definition 3.19. Symmetry: $\frac{E_2=E_1}{E_1=E_2}$

Definition 3.20. Congruence: $\frac{E_1=E_2}{f(E_1)=f(E_2)}$

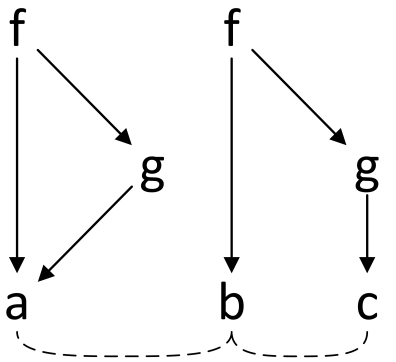
Decision procedures for this theory have great significance, since the decision problem for other theories can be reduced to it. Many theory solvers for uninterpreted functions are based on the *congruence closure* method. If we consider a formula, which consists of conjunctions of equalities between terms using free functions, congruence closure can be applied to find a representation of the smallest set of implied equalities. This is done by converting each term of the formula into a Directed Acyclic Graph (DAG). These DAGs can be used to check if the formula, consisting of a mix of equalities and disequalities, is satisfiable by applying above axioms. Finally, a last check needs to be performed that checks, whether terms on both sides of each disequality are in different equivalence classes [52]. Figure 3.6 shows a step-by-step example of the congruence closure algorithm.

Linear arithmetic

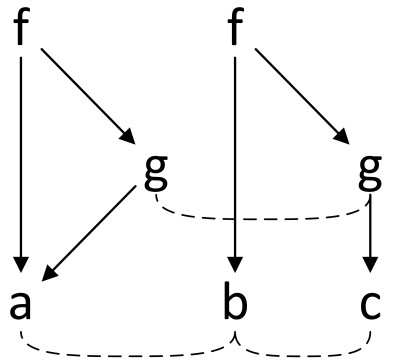
The theory of linear arithmetic is denoted by the SMT-LIB as LIA, LRA, QF_LIA and QF_LRA, which stands for quantified or quantifier-free linear integer arithmetic or linear real arithmetic. Their definitions state that the arithmetical functions add ($+$), subtract ($-$) and multiply ($\cdot$) are supported.



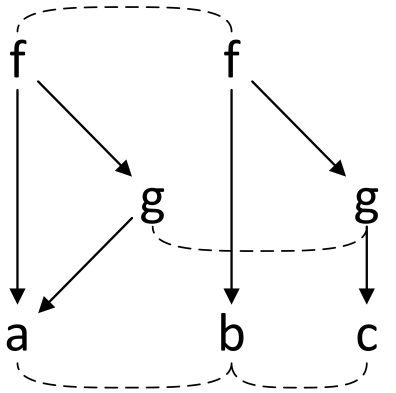
(a) DAGs, each representing a term of the example.



(b) Equivalences $a = b$ and $b = c$ added as dashed lines.

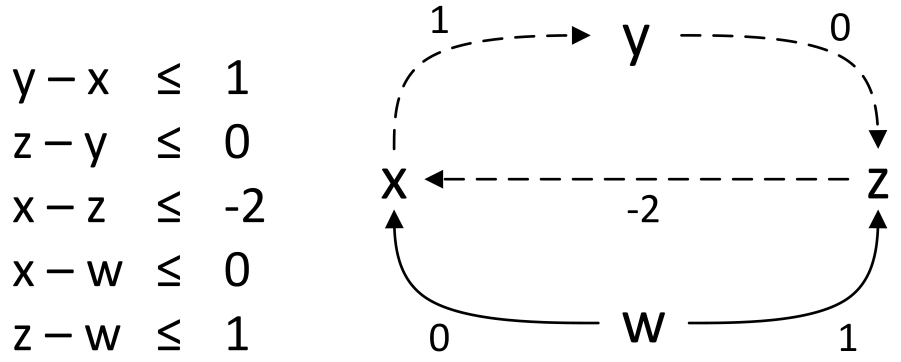


(c) Nodes $g(a)$ and $g(c)$ are congruent because $a = c$, which is implied by the first two equalities (transitivity rule).



(d) Nodes $f(a, g(a))$ and $f(b, g(c))$ are also congruent, since $a = c$ and $g(a) = g(c)$. Therefore, the example is unsatisfiable because the term $f(a, g(a)) \neq f(b, g(c))$ is not true.

Figure 3.6: Congruence closure example: $a = b \wedge b = c \wedge f(a, g(a)) \neq f(b, g(c))$ [52].



(a) An example of conjunctions of difference inequalities.

(b) The example's representation as a graph. The negative cycle is depicted by the dashed lines, making the problem unsatisfiable.

Figure 3.7: Difference inequalities example [52].

However, multiply is restricted to be of form $c \cdot x$, where c is a constant and x a variable. For linear arithmetic over reals the following form of divide (/) is also allowed: c/x, where c is a rational coefficient and x a variable. Furthermore, relational symbols for equality and inequalities ($=$, $\leq$, $<$, ...) are used to form atomic predicates. A popular procedure for deciding linear arithmetic, which many SMT solvers use in their linear arithmetic solver, is called the *simplex* algorithm [52]. Dutertre and de Moura [34] explained in detail how this algorithm works and presented a more efficient version for linear arithmetic solvers for DPLL(T).

Difference arithmetic

The theory of difference arithmetic is denoted by the SMT-LIB as QF_IDL and QF_RDL, which stands for quantifier-free integer difference logic and quantifier-free real difference logic. It is a part of linear arithmetic, where inequalities are restricted to have the form $x - y \leq c$, for variables x, y and constant c. Conjunctions of such inequalities can be solved very efficiently by *searching for negative cycles in weighted directed graphs*. Whereas, each variable represents a node of the graph and an inequality $x - y \leq c$ corresponds to an edge from y to x with weight c [52]. Figure 3.7 shows an example of a conjunction of difference inequalities, as well as its representation as a graph.

Non-linear arithmetic

Non-linear arithmetic is a super-set of linear arithmetic. It is denoted by the SMT-LIB as NIA, NRA, QF_NIA and QF_NRA, which stands for quantified or quantifier-free non-linear integer arithmetic and non-linear real arithmetic. Decision procedures for non-linear arithmetic over reals use algorithms from computer algebra, like *computing a Gröbner basis from equalities* [17]. The problem of deciding satisfiability for non-linear integer arithmetic however, is undecidable. Meaning, there exists no algorithm, which can solve each instance of this problem [52]. Adding quantifiers to the theory makes it even worse. According to [52], there is not even a computable set of axioms for characterizing quantified non-linear integer arithmetic. There are not many SMT solvers that support non-linear arithmetic, which is unfortunate, since we need non-linear real arithmetic to debug spreadsheets.

Bit-vectors

The theory of bit-vectors is denoted by the SMT-LIB as QF_BV: quantifier-free bit-vectors. It represents every number as a fixed-size sequence of bits. In addition to standard arithmetic operations, the theory of bit-vectors also allows mixing bit-wise operations, like NOT, AND, OR, XOR, as well as bit shifts. Efficient decision procedures for bit-vectors use methods, like *lazy bit-blasting* and *approximating long bit-vectors by short bit-vectors* [52].

Arrays

The theory of arrays is denoted by the SMT-LIB as QF_AX: quantifier-free arrays with extensions. As the name suggests, it defines the usage of arrays, which have two special functions:

Definition 3.21. write$(a,\ i,\ v)$: writes value v at index i of array a.

Definition 3.22. read$(a,\ i)$: denotes the value stored in array a at index i.

The definition of the theory of arrays is very vague, to allow for different extensions or restrictions. For instance, some theories restrict, which array sorts are allowed, by restricting its maximal dimension. Furthermore, some could use the theory of array as a basis and with certain extension create for example the theory of lists. The most common approach to deal with

the theory of arrays is to use a reduction to the theory of uninterpreted functions with equality through *lazy array axiom instantiation* [52].

Quantified Theories

If we consider quantifiers part of the language of FOL the problem of deciding satisfiability becomes significantly more difficult. In fact only a few SMT solvers support theories that allow quantifiers. However, if quantifiers are supported, usually some form of *E-matching* is performed to decide satisfiability.

Theory Combination

As already mentioned above, one major difficulty in SMT solver development lies within finding algorithms that not only are able to efficiently handle special theories, but can also be modularly combined with one another. There are several different methods to combine theories for SMT solving that proved themselves in practice, the Nelson-Oppen combination method, the delayed theory combination method and the Model-based theory combination method.

- **Nelson-Oppen Combination [52], [16]:** Assume, we have an SMT input formula φ, of the form:

  $$f(f(x) - f(y)) = a \wedge f(0) = a + 2 \wedge x = y,$$

  as provided by Oliveras and Rodriguez-Carbonell in [55]. To check satisfiability for this formula, we have to combine the theories of uninterpreted functions and linear arithmetic. That is where the Nelson-Oppen combination method comes into play: it purifies the formula φ into $\varphi_1 \wedge \ldots \wedge \varphi_n$ by splitting alphabet Σ, such that, $\varphi_i \in \Sigma_i$. These Σ_i's do not have any common function or predicate symbols, however, they may have shared variables. The purification is done according to the following satisfiability preserving transformation rule:

  $$f(x) \rightarrow f(e) \wedge e = x, \text{where } e \text{ is a fresh variable.}$$

Uninterpreted Functions	**Linear Arithmetic**
$f(e_1) = a$	$e_2 - e_3 = e_1$
$f(x) = e_2$	$e_4 = 0$
$f(y) = e_3$	$e_5 = a + 2$
$f(e_4) = e_5$	
$x = y$	
shared variables: $e_1, e_2, e_3, e_4, e_5, a$	

Table 3.2: Purification example of the Nelson-Oppen combination method.

For our above example, this would split our formula into one part solvable by the theory solver for uninterpreted functions with equality and one part solvable by the linear arithmetic theory solver, as can be seen in Table 3.2.

With that the two theory solvers can check satisfiability for their part of the formula, while propagating entailed equalities of their shared variables between them. This is done until a convergence is reached, meaning, the formula is satisfiable, or until one solver returns unsatisfiable.

- **Delayed Theory Combination:** The delayed theory combination method is a refinement for the Nelson-Oppen method. Instead of directly exchanging equalities between the two theory solvers, the delayed theory combination method takes a different approach. The theory solvers work isolated from each other. All entailed equalities between shared variables are added to both parts of the formula before given to the SAT solver to find a satisfying truth assignment. This assignment is then splitted into different sub-assignments. One assignment for each theory, containing theory pure literals, and one assignment for shared equalities. The later and the corresponding theory assignment is then checked for consistency by each theory solver. If both theory solvers return satisfiable, the formula is satisfiable. Otherwise, the conflict set is added to the formula, to prevent the same truth assignments from occurring. If no T-consistent model can be found, the formula is unsatisfiable [16].

- **Model-based Theory Combination:** The model-based theory combination method also builds upon the Nelson-Oppen combination. For

this approach each theory T_i needs to maintain its own model M_i. When an equality is found, the theory creates a new equality decision literal $(u \simeq v)^d$ and propagates it to all theories sharing u and v. Each of these models M_i need to get changed to satisfy the new literal. In case the equality does not hold within one of the models, satisfiability needs to be checked for the negated literal. If this again leads to an inconsistency, the formula is unsatisfiable. Otherwise, the process is continued until models are found that satisfy the whole formula [29].

3.3.4 Famous Problems expressed as SMT

Let us have a look at a more practical topic. As we already did for CSP and SAT, this section shows how the N-queens puzzle and the map coloring problem can be expressed as an SMT problem.

N-Queens Puzzle

Again we use the 4-queens puzzle and show one possible way of formulating it as an SMT instance.

- **Variables:** $p_{11}, p_{12}, p_{13}, p_{14}, p_{21}, \ldots, p_{43}, p_{44}$ (one variable for each field of the board)
- **Domains:** $D_i = \{0, 1\}$ (for the options: one queen or no queen on a field)
- **Constraints:** no queen can attack another queen

  $(p_{11} + p_{12} + p_{13} + p_{14} = 1) \wedge \ldots$ (for each row and column) $\wedge (p_{11} + p_{22} + p_{33} + p_{44} \leq 1) \wedge \ldots$ (for each diagonal)

Map Coloring Problem

Expressed as an SMT problem the map coloring problem for the map of Austria with three different colors might look like the following:

- **Variables:** W, N, O, ST, B, S, K, T, V (one variable for each territory)
- **Domains:** $D_i = \{1,2,3\}$ (each value represents a color)

- **Constraints:** neighboring territories must have different colors

  $(V \neq T) \wedge (T \neq S) \wedge (T \neq K) \wedge (S \neq O) \wedge (S \neq ST) \wedge (S \neq K) \wedge (K \neq ST) \wedge (ST \neq O) \wedge (ST \neq N) \wedge (ST \neq B) \wedge (O \neq N) \wedge (N \neq W) \wedge (N \neq B)$

3.3.5 Resolution of SMT

As already mentioned before, state-of-the-art SMT solvers use efficient SAT solvers for deciding satisfiability of a formula. However, SAT solvers work on PL and therefore, a conversion from FOL to PL is necessary. Furthermore, PL has a lower expressiveness than FOL and that is why several steps are needed for a successful translation. Once the formula is successfully translated, it can be passed to the SAT solver to decide satisfiability. In modern SMT solvers, there are two common approaches on how the SMT solvers interact with the SAT solver.

Eager approach

SMT solvers that implement the eager approach translate the FOL formula into a PL CNF formula using an algorithm, which preserves satisifability. This is done by considering each atom as a Boolean variable and by adding inconsistencies to the formula. The **eager** approach derives all the inconsistencies before calling the SAT solver. This leads to an easy set-up, since the SAT solver functions as a kind of black-box. However, there might arise the problem that too many inconsistencies get produced, which could turn an easy problem into an impossible one. Most SMT solvers for bit-vectors are based on the **eager** approach, since there exists eager encoding, which prevents the generation of too many inconsistencies [15]. Furthermore, for a correct translation of FOL formulas into PL formulas efficient procedures for every theory are needed. Even though a lot of effort was spend to create algorithms like that, the **lazy** approach is in many cases tremendously faster [54].

The following solving methodology gives a general idea on how an SMT solver, that interacts with its SAT solver according to the eager approach, decides satisfiability for an FOL formula [15]:

- Assume each atom is a Boolean variable.

- Search for all inconsistencies between atoms.
- Translate the formula into a Boolean formula.
- Pass the resulting SAT formula to a SAT solver and return the same result.

Example 3.5. $x = y \wedge (x < y \vee x > y)$
According to above's methodology we first need to consider each atom as a Boolean variable. Therefore, we say $(x = y) \mapsto a$, $(x < y) \mapsto b$ and $(x > y) \mapsto c$. The next step requires us to look for inconsistencies. If $x = y$, neither $x < y$ and $x > y$ can be true. Therefore, our inconsistencies are $\neg(a \wedge b)$ and $\neg(a \wedge c)$. We now translate the FOL formula into a PL formula by converting every atom into a Boolean variable and by adding all found inconsistencies. This leads to the result $a \wedge (b \vee c) \wedge \neg(a \wedge b) \wedge \neg(a \wedge c)$. If this formula is passed to the SAT solver to decide its satisfiability, it would return unsatisfiable. Therefore, the SMT solver's result would also be unsatisfiable, since it returns the same result.

Lazy approach

The **lazy** approach derives inconsistencies during SAT solving. Meaning, it adds inconsistencies on demand and therefore, usually requires less inconsistencies to find a solution. Yet, for the **lazy** approach to work properly, it needs to interface with the SAT solver to decide the T-consistency of the found models. This leads to a more difficult set-up as for the **eager** approach. Nonetheless, the **lazy** approach is, due to its flexibility, the more commonly used approach in existing SMT solvers [15].

Following solving methodology gives a general idea on how an SMT solver, that interacts with its SAT solver according to the lazy approach, decides satisfiability for a FOL formula [15]:

- Assume each atom is a Boolean variable.
- Pass the resulting SAT formula to a SAT solver.
- If the SAT solver returns unsatisfiable return the same result.
- If the SAT solver finds a model, check the model for T-consistency.

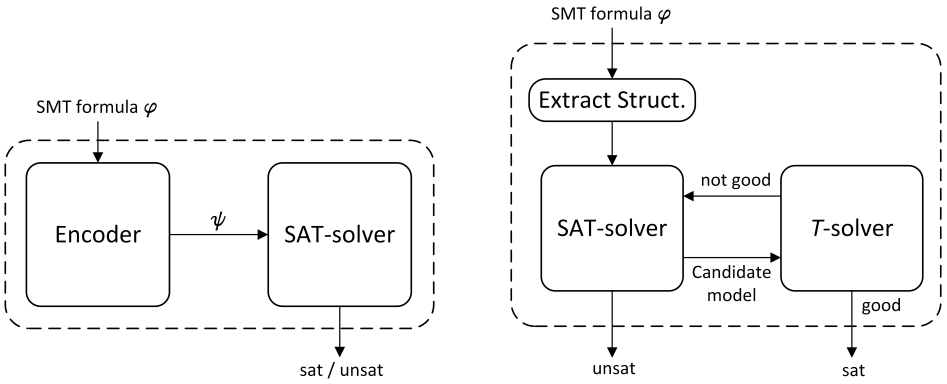
- If the model is T-consistent return satisfiable.
- If the model is T-inconsistent, add theory lemmas to the formula, pass it to the SAT solver and begin anew with deciding its satisfiability.

Example 3.6. $x = y \wedge (x < y \vee x > y)$
Again we have to consider each atom as a Boolean variable: $(x = y) \mapsto a$, $(x < y) \mapsto b$ and $(x > y) \mapsto c$. In the next step we pass the translated formula $(a \wedge (b \vee c))$ to the SAT solver and let it decide satisfiability. In our case the SAT solver would return satisfiable and pass a model to the theory solver. This model could look like $a = 1, b = 1, c = 0$. Meaning, a and b have to be true to make the formula satisfiable. The theory solver checks this model for T-consistency by verifying, if the corresponding FOL atoms can be true as well. In our case a is true and a correlates to $x = y$. Furthermore, b is true, which correlates to $x < y$. For the FOL formula to be satisfiable as well, x and y need to be equal and unequal at the same time, which is not possible. Therefore, the model is T-inconsistent. The theory solver adds this inconsistency, $\neg(a \wedge b)$, to the formula and passes it back to the SAT solver to check for satisfiability. Again the formula would be satisfiable, however, the theory solver would prove the model to be T-inconsistent and add the theory lemma $\neg(a \wedge c)$, leading to the formula being unsatisfiable.

Figure 3.8 shows a high-level view of both the eager and lazy approach.

3.3.6 DPLL(T) Paradigm

Most modern SAT solver are based on the DPLL paradigm, which describes different procedures to efficiently solve SAT problems. SMT solvers make use of this paradigm as well, since they depend on a SAT solver to decide satisfiability. However, as already mentioned, most SMT solvers interact with the SAT solver according to the lazy approach. Therefore, it is necessary to slightly adapt the DPLL paradigm for it to be able to interact with the theory solver and work modulo a theory. A description of these adaptations as well as some commonly used methods, which can greatly enhance the performance of SMT solvers follows below.



(a) **Eager approach:** The encoder adds inconsistencies to the SMT formula and translates it into a propositional formula. The translated formula is then passed to the SAT solver to decide satisfiability.

(b) **Lazy approach:** The formula is translated into a propositional formula and passed to the SAT solver to decide satisfiability. The SAT solver and theory solver interact with each other to decide the T-consistency of the candidate models.

Figure 3.8: Illustration of the eager and lazy approach [15].

Before we can adapt our abstract DPLL model, presented in Section 3.2.4, to work modulo theories, we need to consider that instead of dealing with propositional literals, DPLL(T) deals with quantifier-free first-order ones. Concerning the rules **Decide, Fail, UnitPropagate** and **Restart** that is the only change necessary. As for the rules **Learn, Forget** and **Backjump**, they need to be slightly adapted to work modulo theories. These adaptations are described below and are based on the definitions from [54].

T-Learn

$$M \parallel F \implies M \parallel F,\ C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models_T C \end{cases}$$

Entailment between formulas becomes entailment in T. Also T-learned clauses can belong to M and F, instead of only to F. Otherwise, the rule behaves the same as **Learn** [54].

T-Forget

$$M \parallel F,\ C \implies M \parallel F \quad \textbf{if} \begin{cases} F \models_T C \end{cases}$$

The only change in the **T-Forget** rule is that entailment between formulas becomes entailment in T [54].

T-Backjump

$$M\ l^d\ N \parallel F,\ C \implies M\ l' \parallel F,\ C \quad \textbf{if} \begin{cases} M\ l^d\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F,\ C \models_T C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^d\ N \end{cases}$$

T-Backjump makes use of both, the propositional notion of entailment ($\models$), and the first-order notion of entailment modulo a theory ($\models_T$) [54].

The theory solver waits for the SAT solver to find a model M for the formula. If such a model is found and neither of the rules **Decide, Fail, UnitPropagate** and **T-Backjump** can be applied, the T-solver checks the models consistency. If it is T-consistent, the formula is satisfiable with respect to the theory. Otherwise, if M is T-inconsistent, then there exists a set of literals $\{l_1, \ldots, l_n\}$ in M, which is inconsistent with the theory. **T-Learn** learns the theory lemma $\neg l_1 \vee \ldots \vee \neg l_n$ and **Restart** is applied. This process is repeated until a T-consistent model is found or a *FailState* is reached [54]. Additionally, most modern SMT solvers implement different methods to enhance performance. The most commonly used methods are listed below.

Incremental T-solver

Most state-of-the-art SMT solvers implement the concept of **incremental T-solvers**. This means, instead of waiting for the SAT solver to find a model, the T-consistency of the assignment is checked incrementally while it is being built by the DPLL procedure. This can be done eagerly, that is, detecting T-inconsistencies as soon as they are produced, or in certain intervals, e.g., once every k literals are added to the assignment. For this to work efficiently, the theory solver has to be faster in processing one additional

literal, than in reprocessing the whole set of literals from the beginning. This is, in fact, practicable for many theories but not all [54].

On-line SAT solvers

After a T-inconsistency is detected and learned as a theory lemma, instead of beginning the search anew, the procedure either applies **T-Backjump**, to go back to a point where the assignment was still T-consistent, or produces a *FailState* through the **Fail** rule [54].

Theory propagation

The techniques described up to now allowed the theory solver to only provide information after a T-inconsistent state was reached. **Theory propagation** describes the technique to detect literals l of a formula that are currently true in the partial assignment M (denoted as $M \models_T l$), and adds these literals to M [54]. **Theory propagation** is a kind of forward checking and plays an important role in DPLL(T). Therefore, another rule is added to the abstract model from above.

TheoryPropagate

$$M \parallel F \implies M\ l \parallel F \quad \textbf{if} \begin{cases} M \models_T l \\ l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M \end{cases}$$

TheoryPropagate prunes the search tree by assigning a truth value to T-entailed literals, instead of guessing a value in the **Decide** step.

Exhaustive theory propagation

Exhaustive theory propagation means, to apply theory propagation with a higher priority than the **Decide** rule. Techniques that do not use theory propagation, but instead learn theory lemmas, have to add many consequences of the theory into the clause set and therefore, duplicate the theory information. With **Exhaustive theory propagation** the process of duplicating theory information becomes unnecessary, and non-exhaustive theory propagation reduces it greatly [54].

3.4 Conclusion

After discussing each technology closely in the previous sections let us come to a conclusion. We learned how each solver operates and what kind of problems they are designed to solve. Constraint solvers try through a search and techniques like backtracking and constraint propagation to solve CSPs. SAT solvers solve problems called SAT, which are formulated in the language of PL. Most common solvers are still based on an improved version of the DPLL algorithm, which in its original form is already over 50 years old. SMT solvers work modulo a background theory, and depend upon SAT solvers to help solve their FOL problems, called SMT. There are two different established approaches on how they interact with SAT solvers, the lazy approach and the eager approach. A common procedure to implement SMT solvers is called DPLL(T), on which most state-of-the-art Solver are based on.

Therefore, let us say that both SAT- and SMT solvers are certain forms of constraint solvers, with each focusing on solving different problems. In terms of expressiveness SAT- and SMT solvers are both limited by their languages, which is not true for constraint solvers. Furthermore, constraint solvers can solve more difficult problems, which may not be expressible in PL or FOL.

Chapter 4

SMT Solver Comparison

As already mentioned in Chapter 1, Z3 solves SDP on average six times faster than Choco and Minion—as shown by Außerlechner et al. [6]. Therefore, in this chapter we compare six state-of-the-art SMT solvers concerning their functionality. Furthermore, we list which solvers can be used for MBSD of spreadsheets, and therefore, find suitable candidates to integrate into the framework introduced in Chapter 5. Only solvers that can handle real numbers are considered, since they should be able to debug spreadsheets, and usually spreadsheets contain real numbers. Since the framework uses a solving algorithm called MCSes-U algorithm that makes use of the solvers' functionality to extract unsatisfiable cores, it is important that the solvers support this feature. The following pages give a detailed description of each solver's functionality and technical features as well as a summary of our findings.

Table 4.1 lists all SMT solvers that support real numbers and participated in at least one of the SMT-COMPs from 2005 to 2014.

A summarized overview of the SMT solvers supporting real numbers can be found in Table 4.2 and Table 4.3, whereas, a more detailed description follows below.

Name	SMT-COMP
Barcelogic	2006-2009
CVC/CVCLite/CVC3	2005-2012
CVC4	2010-2014
MathSAT	2005-2014
SMTInterpol	2011-2014
test_pmathsat	2010
veriT	2009-2011, 2013, 2014
Yices 2	2005-2009, 2014
Z3	2007, 2008, 2011, 2013, 2014

Table 4.1: SMT solvers that support real numbers and participated in the SMT-COMP at least once [64], [8], [24]. Solvers shaded in gray are outdated. Either they are no longer in development or a newer version is available.

4.1 Z3

Z3 is a high-performance theorem prover implemented in C++ and developed at Microsoft Research. It is published under the Microsoft Research License Agreement (MSR-LA) license. Application Programming Interfaces (APIs) are available in C, C++, .NET, Python, Java and OCaml. As input language, Z3 supports an extended version of the SMT-LIB v2.0 script language, the Simplify format and the DIMACS format. Furthermore, it is one of the few solvers able to handle every SMT-LIB logic [59]. Z3 participated in many SMT-COMPs over the years and always did very well. In 2011's competition Z3 won QF_BV, QF_UF, QF_LIA, QF_LRA, QF_UFLIA, QF_UFLRA, QF_AUFLIA, QF_IDL, AUFLIA, AUFNIRA among others. Furthermore, 15 benchmarks could only be solved by Z3 and no other solver. A year later Z3 did not participate in the SMT-COMP, yet, the winning solvers could not improve over Z3's 2011 submission in any division, with exception to the division QF_BV [24]. The number of benchmarks that could only be solved by Z3 increased to 21 benchmarks in 2013 [64]. In 2014's SMT-COMP Z3 participated non-competitive, as a reference for the other competitors. However, it still won in 15 divisions out of 32 [23].

Name	Affiliation	Coding Language	License	API	Input Language	Models	Proofs	Unsat-Cores
CVC4 [40]	NYU, U. Iowa	C++	BSD	C++	SMT-LIB v1.0/v2.0, native language	yes	yes	no
MathSAT 5 [20]	U. Trento, FBK-irst	C++	Proprietary	C	SMT-LIB v1.2/v2.0, DIMACS format, native language	yes	yes	yes
SMTInterpol [19]	U. Freiburg	Java	LGPL v3	Java	SMT-LIB v1.2/v2.0, DIMACS format	yes	yes	yes
veriT [32]	U. Nancy, INRIA, UFRN	C	BSD	C	SMT-LIB v2.0, DIMACS format	yes	yes	no
Yices 2 [33]	SRI	C	Proprietary	C	SMT-LIB v1.2/v2.0, native language	yes	no	no
Z3 [31]	Microsoft Research	C++	MSR-LA	C, C++, .NET, Python, Java, OCaml	SMT-LIB v2.0, Simplify format, DIMACS format	yes	yes	yes

Table 4.2: Overview of SMT solvers supporting real numbers, regarding their affiliation, code basis, provided interfaces and supported functionality. Columns shaded in light gray denote the necessary functionality for spreadsheet debugging. Columns denoting the functionality required to work in combination with the MCSes-U algorithm [48] are shaded in dark gray.

Name	QF_UF	QF_AX	QF_BV	QF_DL	QF_LA	QF_NA	Quantifiers
CVC4	yes	yes	yes	yes	yes	yes	yes
MathSAT 5	yes	yes	yes	no	yes	no	no
SMTInterpol	yes	no	no	no	yes	no	no
veriT	yes	no	no	yes	yes	no	yes
Yices 2	yes	yes	yes	yes	yes	no	no
Z3	yes	yes	yes	yes	yes	yes	yes

Table 4.3: Overview of supported theories. Columns for arithmetic theories always apply for integer and real arithmetic. Not listed in this table are the supported combinations of the theories. They can be found in the individual solver's description. The columns shaded in light gray denote the solvers which support the theories relevant for value-based spreadsheet debugging.

Technical characteristics

A more detailed description of Z3's characteristic features follows below [31], [30].

- **User Interaction:** Z3 supports many different input formats. Next to its own native input language, Z3 accepts the SMT-LIB v2.0 script language, the Simplify format and the DIMACS format as input. Furthermore, it is also accessible via an API, which is available in C, C++, .NET, Python, Java and OCaml.

- **Simplifier:** Z3 implements a module called simplifier. The simplifier simplifies the input formulas by applying standard algebraic reduction rules and contextual simplifications.

- **Core technology:** Like most modern SMT solvers, Z3 is based on the lazy/DPLL(T) paradigm. Furthermore, it integrates a custom DPLL-based SAT solver with functionalities like standard search pruning methods, two-watch literals for constraint propagation, lemma learning using conflict clauses, and non-chronological backtracking.

- **Theory solver:** Z3 integrates a core theory solver that handles equalities and uninterpreted functions and different satellite solvers for linear arithmetic, bit-vectors, arrays and others. Additionally, Z3 makes use of an E-matching machine to handle quantifiers. The theory solver for uninterpreted functions with equalities is based on the congruence closure algorithm. A basis for the linear arithmetic theory solver provides the simplex algorithm. The theory solver for arrays uses lazy instantiation of array axioms. Whereas, the bit-vector theory solver applies bit-blasting to all bit-vector operations except equality. To combine theories, Z3 makes use of the model-based theory combination method [10].

- **Models and Proofs:** Microsoft's SMT solver is able to return models for satisfiable formulas and generate proofs and extract unsatisfiable cores for unsatisfiable formulas. An unsatisfiable core is a set of clauses which are T-unsatisfiable. Z3, SMTInterpol and MathSAT 5 are currently the only three SMT solvers supporting unsatisfiable core extraction.

4.2 CVC4

CVC4 stands for *Cooperating Validity Checker* and is a joint project of New York University and University of Iowa. It is an open-source software written in C++ and published under the terms of the modified Berkeley Software Distribution (BSD) license. Although, some builds link against libraries published under the GNU General Public License (GPL) and therefore, the use of these builds is allowed for open-source projects only. CVC4 accepts three different input languages, namely SMT-LIB v1.0, SMT-LIB v2.0 and CVC4's own native language. Furthermore, CVC4, like Z3, is able to handle every SMT-LIB logic [40]. Like MathSAT 5, CVC4 and its predecessors participated in every SMT-COMP from 2005 to 2014. CVC4 is one of the few SMT solvers that support quantifiers. In 2012, CVC4 and its predecessor CVC3 were the only submissions for the divisions including quantifiers. That is why they were only run as a demonstration. The result showed though that neither of the two did improve over Z3, the winner of the year before. For the theory of QF_UFLRA CVC4 won against four other participants, and managed to improve over one but not all of the winners in 2011 [24]. In 2014's SMT-COMP CVC4 won in the divisions AUFLIA, AUFNIRA, LRA, QF_AUFBV, QF_LIA, QF_LRA, QF_UFNIA, UF and UFLIA [23].

Technical characteristics

CVC4 has four predecessors to learn from in terms of implementation architecture, and efficiency. Therefore, it offers many different features [40], [7].

- **User Interaction:** As already mentioned, CVC4 supports SMT-LIB language v1.0 and v2.0, as well as its own native language. It reads input from an external file and recognizes the input language by the file's extension (.cvc for CVC4's native language, .smt2 for SMT-LIB v2.0 and .smt for SMT-LIB v1.0). Additionally, the user can specify the type of the input language by a command line flag. CVC4 is also accessible via a C++ API.
- **Core technology:** CVC4 is based on the lazy/DPLL(T) approach. As for satisfiability checking, CVC4 theoretically allows for different SAT solvers to be plugged in, yet, up till now, only MiniSAT is supported.

- **Theory solver:** CVC4 uses approaches based on the modern DPLL(T) paradigm, implementing different theory solvers for each theory. The theory solver for uninterpreted functions is based on the congruence closure algorithm. In case of the theory solver for linear arithmetic, the implementation is based on the simplex method. The array theory solver makes us of lazy instantiation of array axioms and the approach used for the theory of bit-vectors combines lazy bit-blasting with in-processing using an algebraic solver. For combining theories, the solver relies on polite combination and care functions.

- **Models and Proofs:** CVC4 has functionalities for generating models and proofs. Its proof system is designed to have absolutely zero footprint in memory and time, when switched off at compile-time. It also supports the Logical Framework with Side Conditions (LFSC), which is a high-level declarative language for defining proof systems and proof objects for almost any logic. LFSC supports computational side conditions on proof rules, which facilitate the design of proof systems [60]. Unfortunately, CVC4 does not yet support unsatisfiable core extraction.

- **Parallel solving:** CVC4 provides an opportunity to run multiple instances of CVC4 in different threads. Although, lemmas are not shared between threads by default, there exists an option to do so. With this option switched on, CVC4 is able to share lemmas of n literals, excluding literals that are local to one thread and therefore, ineligible for sharing. Operations can be interrupted, if results from another thread make them irrelevant. Even though this is a great feature, it is still in an experimental state and thus, should be used with caution.

4.3 MathSAT 5

MathSAT 5 is a joint project of Fondazione Bruno Kessler (FBK-irst) and University of Trento. It is implemented in C++ and freely available for academic research and evaluation purposes. MathSAT 5's default input format is SMT-LIB v2.0. Additionally, MathSAT 5 supports SMT-LIB v1.2 or the DIMACS format. It supports most of the SMT-LIB logics, like that of

equality and uninterpreted functions (QF_UF), linear arithmetic over integers and reals (QF_LIA, QF_LRA), arrays (QF_AX), bitvectors (QF_BV) and floating point numbers, as well as their combinations (QF_UFLIA, QF_UFLRA, QF_AUFBV, QF_AUFLIA, QF_UFBV) [20]. Like CVC4, MathSAT 5 and its predecessors have participated in every SMT-COMP taken place from 2005 to 2014. In 2010, MathSAT 5 won in the divisions QF_UFLRA and QF_UFLIA. Two years later in 2012, MathSAT 5 again won QF_UFLIA [8]. MathSAT 5, like SMTInterpol, was one of only 2 participants in the *unsat core track* of 2012's SMT-COMP [24]. In 2014, MathSAT 5, like Z3, was a non-competitive participant in the SMT-COMP. Unlike Z3, MathSAT 5 did not win any division. However, it did perform very well, considering that it was not optimized for any of the benchmarks [23].

Technical characteristics

MathSAT 5 has been in constant development for many years to provide a vast array of functionality for its users [20].

- **User Interaction:** Users can interact with MathSAT 5 via command line, by providing an SMT-LIB script file, in either v1.2 or v2.0. The solver also accepts input in the form of the DIMACS format. Furthermore, MathSAT 5 provides a C API, which is similar to the commands of the SMT-LIB v2.0 language.
- **CNF Converter:** MathSAT 5's constraint encoder converts every input formula into its CNF.
- **Core technology:** By default, MathSAT 5's core consists of a MiniSAT-style SAT solver, which interacts with the theory solvers according to the lazy/DPLL(T) procedure.
- **Theory solver:** MathSAT 5 consists of individual theory solvers based on state-of-the-art algorithms. The solver for uninterpreted functions is based on the congruence closure algorithm. As for the linear arithmetic on integers and rationals a layered approach based on simplex and branch & bound is used. For the floating point theory, MathSAT 5 implements two different approaches. One is based on either lazy or eager bit-blasting. The second and more recent one is

based on a combination of Interval Constraint Propagation for floating point numbers and modern CDCL SAT solvers. For the array theory solver MathSAT 5 uses axiom instantiation. The bit-vector theory solver uses either lazy or eager bit-blasting and the combination of theories is handled by MathSAT 5's delayed theory combination framework.

- **Models and Proofs:** In addition to deciding satisfiability, MathSAT 5 is able to enumerate models with different truth values for satisfiable formulas, or a resolution proof and theory specific sub-proofs of the T-lemmas for unsatisfiable formulas. Furthermore, it can extract unsatisfiable cores or Craig interpolants [21].

  Craig's interpolation theorem describes a certain relationship between two logical formulas. Lately, this theorem was introduced into the world of SMT and soon became very popular. If we consider an SMT problem for the background theory T and an ordered pair of formulas (A, B), such that, A and B are unsatisfiable under the theory T ($A \wedge B \models_T \bot$), then a Craig interpolant is a formula I for which holds: [21]

  - A satisfies I under the theory T: $A \models_T I$,
  - I and B are unsatisfiable under the theory T: $I \wedge B \models_T \bot$,
  - I precedes, or is the same as A and I precedes, or is the same as B: $I \preceq A$ and $I \preceq B$

- ***AllSMT* and Predicate Abstraction:** MathSAT 5 implements an *AllSMT* functionality. Meaning, for a satisfiable formula, it can efficiently generate a complete set of theory-consistent partial assignments satisfying the formula.

- **Pluggable SAT solvers:** MathSAT 5 allows its users to integrate an external SAT solver of their choice.

4.4 SMTInterpol

SMTInterpol is an interpolation SMT solver developed by the University of Freiburg. It is implemented in Java and available under the open source software license GNU Lesser General Public License (LGPL) v3. As input language SMTInterpol supports SMT-LIB v1.2 and v2.0, as well as the DIMACS format. The solver supports the theories of uninterpreted functions with equality (QF_UF), linear arithmetic over integers and reals (QF_LIA, QF_LRA) and the combination of these theories (QF_UFLIA, QF_UFLRA). SMTInterpol also participated in the SMT-COMP of 2011 in both the *main* and the *application track* and was able to solve as many problems as the winning solver, but with an inferior runtime [19]. In 2012's SMT-COMP SMTInterpol was open-source winner for the theory of QF_UFLIA and sole competitor in the *proof generation track*. Furthermore, SMTInterpol and MathSAT 5 were the only two solvers that participated in 2012's *unsat core track*. Meaning, in the field of proof generation and unsat core extraction, SMTInterpol is one of leading SMT solvers available [24].

Technical characteristics

SMTInterpol is a very "young" SMT solver. However, it provides a wide range of features for its users [19].

- **User Interaction:** SMTInterpol is SMT-LIB v1.2/v2.0 compliant. Meaning, it supports the SMT-LIB script language. Furthermore, it includes a parser for the DIMACS format. It also provides a Java API modeled after the commands of this language. Users can issue commands through an SMT-LIB file, use the standard input channel of the solver, or use the Java API.
- **CNF Converter:** SMTInterpol converts every input formula into its CNF.
- **Core technology:** SMTInterpol is based on the DPLL(T) paradigm and interacts with its SAT solver according to the lazy approach. The implemented SAT solver is a CDCL engine. Meaning, the SAT solver is based on the DPLL algorithm, but with the one difference that its backtracking is non-chronologically.

- **Theory solver:** SMTInterpol consists of two theory solvers, one for uninterpreted functions, which is based on the congruence closure algorithm, and one for linear arithmetic based on the simplex algorithm. Furthermore, it uses the model-based theory combination procedure to combine theories.
- **Models and Proofs:** The solver can return models for formulas which are satisfiable. For unsatisfiable formulas it can produce resolution proofs from which it can extract unsatisfiable cores or Craig interpolants.

4.5 veriT

VeriT was created in a joint work of the University of Nancy, Institut national de recherche en informatique et en automatique (INRIA) and Federal University of Rio Grande do Norte (UFRN). It is an open-source tool written in C and distributed under the BSD license. VeriT supports SMT-LIB v2.0 and DIMACS as a valid input format [12]. Up to 2011 veriT provided a decision procedure for the logic of quantifier-free formulas over uninterpreted symbols (QF_UF), difference logic over integer and real numbers (QF_IDL, QF_RDL), and the combination thereof (QF_UFIDL). Since then the program has been completely rewritten to also support linear arithmetic and quantifier reasoning capabilities. In 2014' SMT-COMP veriT participated in the divisions QF_IDL, QF_LIA, QF_LRA, QF_RDL, QF_UF and in the combinations thereof QF_AUFLIA, QF_UFIDL, QF_UFLIA, QF_-UFLRA, as well as in the divisions allowing quantifiers ALIA, AUFLIA, AUFLIRA, LIA, LRA, UF, UFLIA, UFLRA [32]. VeriT has yet to win a division of an SMT-COMP, but that has low significance, since veriT is still in its early tracks.

Technical characteristics

VeriT's different features are discussed below [12].

- **User Interaction:** VeriT is compliant to SMT-LIB v2.0. In case, one wants to use veriT as a SAT solver, the DIMACS format should be the input language of choice. Of course, veriT is also accessible via a C API.

- **Core technology:** The basis for the solver builds the lazy/DPLL(T) approach. As integrated SAT solver, veriT makes use of MiniSAT.
- **Theory solver:** VeriT's reasoning engine for linear arithmetic is based on the Simplex method. The solver handling uninterpreted functions is based on the congruence closure method. Furthermore, veriT integrates some level of quantifier reasoning through E-matching. To combine theories veriT makes use of the Nelson-Oppen theory combination method [32].
- **Models and Proofs:** The prover uses the MiniSAT solver to produce models for the Boolean abstaction of the input formula. It can also produce proof traces for quantifier-free formulas containing uninterpreted functions and arithmetic. Unfortunately, veriT does not yet support unsatisfiable core extraction.

4.6 Yices 2

Yices 2 is an efficient SMT solver developed by the Stanford Research Institute (SRI International). It is a closed-source software written in C and distributed free-of-charge for personal use under the terms of the Yices license. Yices 2 can decide satisfiability for formulas consisting of quantifier-free combinations of uninterpreted functions with equality (QF_UF), linear arithmetic over integers and reals (QF_LIA, QF_LRA), bit-vectors (QF_-BV), arrays (QF_AX) and integer and real difference logic (QF_IDL, QF_-RDL) and (QF_UFLIA, QF_UFLRA, QF_AUFBV, QF_AUFLIA, QF_-UFBV QF_UFIDL). These are all SMT-LIB logics, which do not involve quantifiers and nonlinear arithmetic. Additionally, it supports tuples and enumeration types. As input, Yices 2 supports SMT-LIB v1.2 and SMT-LIB v2.0 syntax, as well as its own specification language [33]. Yices 2 and its predecessor have participated in the SMT-COMPs of 2005 to 2009 and again in 2014. Furthermore, Yices 2 defeated Z3 in 2008, in the divisions QF_UF and QF_LRA. In 2009, Yices 2 won in the divisions QF_AX, QF_-UFLRA, QF_AUFLIA, QF_UFLIA, QF_UF, QF_RDL and QF_LRA [8]. In 2014's SMT-COMP Yices 2 won the divisions QF_ALIA, QF_AUFLIA, QF_AX, QF_RDL, QF_UF and QF_UFBV [23].

Technical characteristics

To see what sets Yices 2 apart from the other solvers, its specifics are listed below [33].

- **User Interaction:** Yices 2 can read and process input in the form of SMT-LIB notation, in either v1.2 or v2.0, as well as in its own specification language. Furthermore, it is accessible via its C API.
- **Core technology:** Since Yices 2 is closed-source, one can only guess the technology behind, although it is known that it is based on the lazy/DPLL(T) approach and its custom SAT solver is based on the CDCL procedure.
- **Theory solver:** Yices 2 currently implements four different theory solvers configured for uninterpreted functions, linear arithmetic, arrays and bit-vectors, respectively. It is possible to manually couple these components with the SAT solver or remove them individually, if not needed, to optimize runtime for specific problems. The solver for uninterpreted functions is based on the congruence closure method. Linear arithmetic theories are handled by the theory solver based on the simplex algorithm. The decision procedure for the theory of arrays uses lazy instantiation of array axioms. Bit-vectors are handled using bit-blasting. To combine theories, Yices 2 makes use of the Nelson-Oppen theory combination method.
- **Models but no Proofs:** Yices 2's API provides functions to create models, which map the formula's symbols to concrete values. Unfortunately, it does not support commands to get proofs or unsatisfiable cores.

4.7 Findings

The six previously described SMT solvers are all able to handle real numbers. Furthermore, each solver provides functionality to produce models for satisfiable formulas and with the exception of Yices 2 all solvers are able to generate proofs for unsatisfiable formulas. The extraction of unsatisfiable cores is supported only by MathSAT 5, SMTInterpol and Z3. Non-linear

arithmetic support is provided only by Z3. CVC4 supports non-linear arithmetic in a very restricted form, by converting non-linear arithmetic expressions into linear arithmetic. However, our tests showed that this approach to handle non-linear arithmetic is not suited for spreadsheet debugging, since the solver cannot even solve very simple spreadsheet debugging problems. This leads us to the conclusion that Z3 is currently the only suitable candidate for MBSD of spreadsheets.

Chapter 5

Framework and Implementation

Recently, a team at the Graz University of Technology developed a framework [6] to compare the performance and execution time of different SMT- and constraint solvers when debugging spreadsheets. The framework was developed to find an efficient way to debug spreadsheets and thus develop a tool for end-users to verify spreadsheets. Within our work we extend the framework with new model creation approaches for model-based spreadsheet debugging with SMT solvers. Therefore, in this chapter we describe the framework's existing functionality and design. Furthermore, we introduce new model creation approaches and how they can be integrated into the framework.

5.1 Existing

In this section we discuss the framework's existing functionality important for model-based spreadsheet debugging with SMT solvers. Currently, there are three different solvers integrated in the framework, two constraint solvers, called Minion and Choco and one SMT solver, called Z3. Furthermore, the framework includes different constraint solving methodologies. Figure 5.1 shows an overview of the framework's basic components. The components shaded in gray depict the ones relevant for spreadsheet debugging with SMT solvers. These are the components we focus on within this section. We give an overview of how the framework creates value-based

models for Z3 as well as describe its different solving methodologies. Furthermore, we give a list of supported standard spreadsheet functions, for which the framework is able to produce Z3 constraints.

The framework takes a spreadsheet—as defined in Chapter 2—and a corresponding property file as input. A property file can be considered as a failing test case. It is composed of a list of the spreadsheet's correct output cells and the incorrect output cells with their expected values. However, we defined a test case in Definition 2.12 to also contain the spreadsheet's input cells. Since the property file does not contain a list of input cells, the framework extracts the list from the spreadsheet itself. Furthermore, the property file contains a list of the faulty cells (cells causing an error), only for testing purposes, to automatically check whether the right solutions are found. In addition to spreadsheets and property files the framework accepts user input, to choose a solver and since the framework integrates two different solving algorithms for Z3, the user has to specify which algorithm should be used. Once a solver is chosen—in our case that is Z3—the framework creates a SDP by converting the spreadsheet and the data from the corresponding property file into a value-based model. The model is specifically created for the chosen solver by accessing its API. Finally, one of the solving algorithms is executed which enumerates all diagnoses for the faulty spreadsheet by solving the SDP. That is only a short overview of how the framework debugs spreadsheets with Z3, a detailed description of value-based models and the different solving algorithms follows.

5.1.1 Model-based Software Debugging

To create an SDP using Model-based Software Debugging (MBSD) we need a spreadsheet Π, as defined in Chapter 2 and some given observations, f.i. a failing test case T, as defined in Definition 2.13. Furthermore, we need to create a model M, which represents the spreadsheet as a constraint system. This model is used to detect contradictions between the expected output cells defined in T and the obtained output cells from Π [51]. To decide which cells could resolve these discrepancies, MBSD introduces so called not-abnormal variables (NAB). These not-abnormal variables represent the "health" state of a cell. In case a cell c is not-abnormal the formula of the cell

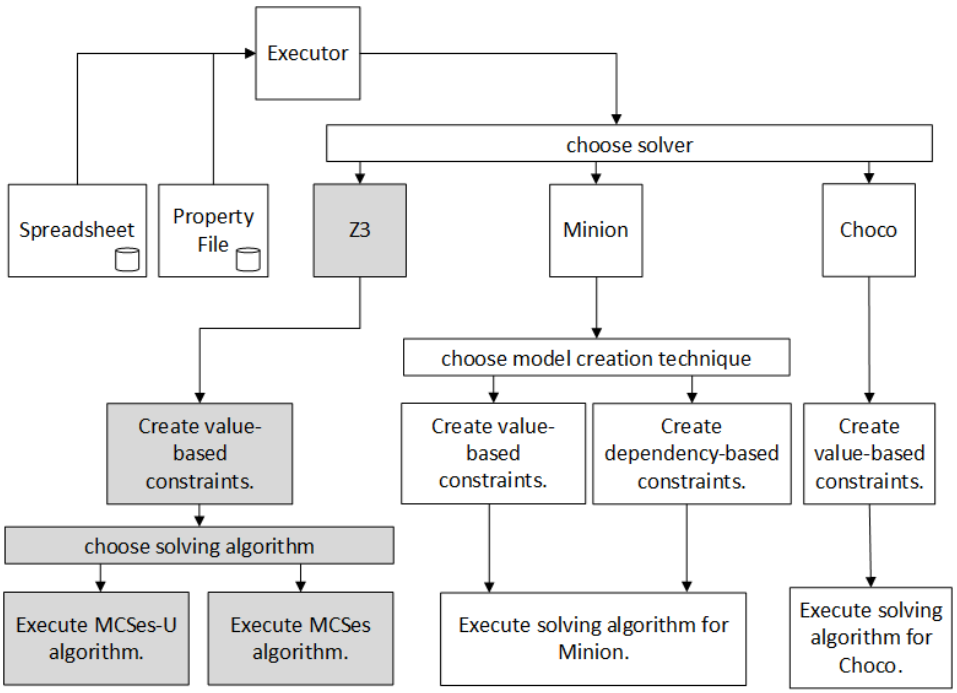


Figure 5.1: A general overview of the framework's components. Components shaded in gray depict the ones relevant for spreadsheet debugging with SMT solvers.

must be correct, otherwise it is suspected to be faulty. As Hofer et al. [43] described this can be represented as follows:

$$\text{NAB}(\text{cell}_c) \rightarrow \mathit{constraint}(c).$$

If we pass a constraint system like this to a constraint or SMT solver, it can compute which not-abnormals have to be set to *false* in order to get rid of the discrepancies. MBSD can be utilized with different kinds of models. For Z3 the framework makes use of value-based models.

Value-based Models

Like Hofer et al. [43] describe, value-based models take the cells' values into account. This leads to following constraint generation rules for value-based models:

$$\forall \text{ cells } c \in \mathit{Input}(\Pi) : c == v(c)$$

$$\forall \text{ cells } c \in \mathit{Output}(\Pi) : c == v_{exp}(c)$$

$$\forall \text{ cells } c \in \bigcup_{c' \in O_{wrong}} \textsc{Cone}(c') : \text{NAB}(\text{cell}_c) \rightarrow c == \mathit{constraint}(f(c))$$

$$\forall \text{ cells } c \in \Big(\bigcup_{c' \in O_{correct}} \textsc{Cone}(c') \setminus \bigcup_{c'' \in O_{wrong}} \textsc{Cone}(c'') \Big) : c == \mathit{constraint}(f(c))$$

with functions $\mathit{Input}(\Pi)$, $\mathit{Output}(\Pi)$ and $\textsc{Cone}(c)$ defined as in Definitions 2.10, 2.11 and 2.15 and O_{wrong} and $O_{correct}$ defined as in Definition 2.13. Since formulas represented as constraints are handled as equations instead of assignments, it is possible to draw conclusions not only from the input on the output but also vice versa.

Generating Value-based Models

To convert spreadsheets into value-based models the framework makes use of Algorithm 5.1. It is a modified version of the algorithm presented by Abreu

et al. [4]. The algorithm requires a spreadsheet Π and a failing test case T as input and returns a value-based model representing Π and T. First each cell for which holds:

$$c \in \bigcup_{c' \in O_{wrong}} \textsc{Cone}(c')$$

is added to a set of cells O_w at the Lines 5 to 7. Additionally, each cell for which holds:

$$c \in \bigcup_{c' \in O_{correct}} \textsc{Cone}(c') \backslash O_w$$

is added to a set of cells O_c at the Lines 8 to 10. The formulas of cells which are elements of the set O_w are converted into clauses using Algorithm 5.2 at Line 12. Line 13 generates the cells' Boolean not-abnormal variables and links them with the cells' value-based constraints. The functions $\textsc{Name}(c)$ and $\textsc{Index}(c)$ respectively, return a unique name or index for each cell c. Then the cells' clauses are added to the spreadsheet clause set C_Π at Line 14. The formulas of cells which are elements of the set O_c are converted into clauses by Algorithm 5.2 at Line 17. Line 18 generates the cells' value-based constraints, which are added to the clause set C_Π at Line 19. At Lines 22 to 27 the constraints for the test case information are generated. Finally, the conjunction of the spreadsheet's constraint set C_Π and the test case's constraint set C_T is returned at Line 28.

Algorithm 5.2 converts each cell's formula into a value-based constraint. Lines 2 to 4 state that a constant is represented by itself. Lines 5 to 7 express that referenced cells are represented by their unique name. If an expression e consists of an expression e_1 enclosed in parentheses, it is represented by the clause of e_1 as shown at the Lines 8 to 11. Expressions of the form e_1 o e_2 are handled by the Lines 12 to 18. Each sub-expression (e_1 and e_2) is converted into a clause separately. Additionally, a new clause is created that represents the relationship between e_1, e_2, operator o and auxiliary variable *result*. The supported spreadsheet functions are converted separately into constraints, since they differ in syntax and semantic. As representatives two examples for the conversion of spreadsheet functions into constraints are given. Lines 19 to 26 and Lines 27 to 30 show the representation of conditionals and sums, respectively.

Algorithm 5.1 Algorithm to convert a spreadsheet into a value-based model [4].

Require: Spreadsheet Π, a failing test case T with input I and output O_{wrong} and $O_{correct}$
Ensure: Value-based clause set of Π and T

1: **function** ConvertSpreadsheetIntoVBModel(Π, T)
2: $C_\Pi := \emptyset$
3: $O_w := \emptyset$
4: $O_c := \emptyset$
5: **for all** cells $c \in O_{wrong}$ **do**
6: $O_w := O_w \cup \text{Cone}(c)$
7: **end for**
8: **for all** cells $c \in O_{correct}$ **do**
9: $O_c := O_c \cup \text{Cone}(c) \setminus O_w$
10: **end for**
11: **for all** cells $c \in O_w$ **do**
12: $[C, aux] :=$ ConvertExpressionVB(f(c))
13: $clause :=$ NAB(Index(c)) $\rightarrow$ (Name(c) == aux)
14: $C_\Pi := C_\Pi \wedge C \wedge clause$
15: **end for**
16: **for all** cells $c \in O_c$ **do**
17: $[C, aux] :=$ ConvertExpressionVB(f(c))
18: $clause :=$ Name(c) == aux
19: $C_\Pi := C_\Pi \wedge C \wedge clause$
20: **end for**
21: $C_T := \emptyset$
22: **for all** tuples $(c,v) \in I$ **do**
23: $C_T := C_T \wedge (\text{Name}(c) == v)$
24: **end for**
25: **for all** tuples $(c, v_{exp}) \in O$ **do**
26: $C_T := C_T \wedge (\text{Name}(c) == v_{exp})$
27: **end for**
28: **return** $C_\Pi \wedge C_T$
29: **end function**

Definition 5.1. Clause for conditionals: $\Psi(cond,\ e_1,\ e_2,\ result)$ is a clause, which ensures the relationship of *cond*, e_1, e_2 and *result* like follows: If *cond* is true *result* = e_1 else *result* = e_2.

Definition 5.2. Clause for sums: $\textsc{Sum}(c_1 \ldots c_2,\ result)$ is a clause, which ensures that *result* equals the sum of the values contained in the area c_1:c_2.

Running Example

To give a demonstration of how value-based models can look like, we introduce a running example, as shown in Figure 5.2. Figure 5.2a shows the value view of the correct version of the example spreadsheet. The spreadsheet shows the moving behavior of an object in three different phases. The first phase describes a constant acceleration of the object which is calculated in column B. In phase two the object moves with constant velocity, calculated in column C. Column D calculates the third phase which describes a constant deceleration of the moving object until it stops because its velocity reaches zero, as shown in Cell E2. For better demonstration we shaded the input cells of the spreadsheet in gray and the correct output cells in green. Figure 5.2b shows the value view of the faulty spreadsheet. We manually injected a fault in Cell C5—colored in red—where we purposely used the initial velocity of phase one (Cell B2) instead of phase two (Cell C2). This can be seen in Figure 5.2c, which shows the formula representation of the faulty spreadsheet. The cells colored in yellow depict the output cells which are wrong due to their references to the faulty cell. Table 5.1 shows the failing test case for the faulty spreadsheet from Figure 5.2b. It contains the input cells with their values and the output cells with their expected values.

Example 5.1. If the faulty spreadsheet from Figure 5.2b and the failing test case from Table 5.1 are passed to Algorithm 5.1, we receive a constraint set as shown in Table 5.2. Solving this constraint set results in a single-fault diagnosis stating that Cell C5 has to be a faulty cell.

5.1.2 Z3's Solving Methodologies

For Z3 the framework integrates two different solving methodologies. They are called the MinimalCorrectionSets algorithm (Algorithm 5.3) and

Algorithm 5.2 Algorithm to convert an expression into a value-based constraint [4].

Require: Expression e
Ensure: $[C, var]$ with C as a clause, and var as the name of an auxiliary variable, a cell name or a constant

```
1:  function ConvertExpressionVB(e)
2:      if e is a constant then
3:          return [∅, e]
4:      end if
5:      if e is a cell name then
6:          return [∅, Name(e)]
7:      end if
8:      if e is of the form (e1) then
9:          [C, aux] := ConvertExpressionVB(e1)
10:         return [C, aux]
11:     end if
12:     if e is of the form e1 o e2 then
13:         [C1, aux1] := ConvertExpressionVB(e1)
14:         [C2, aux2] := ConvertExpressionVB(e2)
15:         var result
16:         Create a new clause clause according to the given operator o,
            which defines the relationship between aux1, aux2 and result
17:         return [C1 ∧ C2 ∧ {clause}, result]
18:     end if
19:     if e is of the form IF(cond,e1,e2) then
20:         [C1, aux1] := ConvertExpressionVB(cond)
21:         [C2, aux2] := ConvertExpressionVB(e1)
22:         [C3, aux3] := ConvertExpressionVB(e2)
23:         var result
24:         C := C1 ∧ C2 ∧ C3
25:         return [C ∧ Ψ(aux1, aux2, aux3, result), result]
26:     end if
27:     if e is of the form SUM(c1:c2) then
28:         var result
29:         return [Sum(c1 ... c2, result), result]
30:     end if
31:     ...
32: end function
```

	A	B	C	D	E
1		Constant Acceleration	Constant Velocity	Constant Deceleration	Final State
2	Initial Velocity [m/s]	0,0	20,0	20,0	0,0
3	Acceleration [m/s^2]	2,0	0,0	-4,0	
4	Duration [s]	10,0	10.000,0	5,0	
5	Distance [m]	100,0	200.000,0	50,0	
6	Accumulated Distance [m]	100,0	200.100,0	200.150,0	

(a) Correct spreadsheet

	A	B	C	D	E
1		Constant Acceleration	Constant Velocity	Constant Deceleration	Final State
2	Initial Velocity [m/s]	0,0	20,0	20,0	0,0
3	Acceleration [m/s^2]	2,0	0,0	-4,0	
4	Duration [s]	10,0	10.000,0	5,0	
5	Distance [m]	100,0	0,0	50,0	
6	Accumulated Distance [m]	100,0	100,0	150,0	

(b) Faulty spreadsheet

	A	B	C	D	E
1		Constant Acceleration	Constant Velocity	Constant Deceleration	Final State
2	Initial Velocity [m/s]	0	=B2+B3*B4	=C2+C3*C4	=D2+D3*D4
3	Acceleration [m/s^2]	2	0	-4	
4	Duration [s]	10	10000	5	
5	Distance [m]	=B2*B4+B3*B4*B4/2	=B2*C4+C3*C4*C4/2	=D2*D4+D3*D4*D4/2	
6	Accumulated Distance [m]	=B5	=B5+C5	=B5+C5+D5	

should be: C2*C4+C3*C4*C4/2

(c) Formula view of the faulty spreadsheet

Figure 5.2: Running example

Input cells with values:	**Output cells with expected values:**
B2 == 0	B6 == 100
B3 == 2	C6 == 200,100
B4 == 10	D6 == 200,150
C3 == 0	E2 == 0
C4 == 10,000	
D3 == -4	
D4 == 50	

Table 5.1: The failing test case for the running example from Figure 5.2b.

Input Cells:	**Output Cells:**
B2 == 0	B6 == 100
B3 == 2	C6 == 200,100
B4 == 10	D6 == 200,150
...	E2 == 0
Formula Constraints:	
NAB(B5) $\rightarrow$ B5 == B2 $\cdot$ B4 + B3 $\cdot$ B4 $\cdot$ B4 / 2	
NAB(C2) $\rightarrow$ C2 == B2 + B3 $\cdot$ B4	
NAB(C5) $\rightarrow$ C5 == B2 $\cdot$ C4 + C3 $\cdot$ C4 $\cdot$ C4 / 2	
NAB(C6) $\rightarrow$ C6 == B5 + C5	
NAB(D2) $\rightarrow$ D2 == C2 + C3 $\cdot$ C4	
NAB(D5) $\rightarrow$ D5 == D2 $\cdot$ D4 + D3 $\cdot$ D4 $\cdot$ D4 / 2	
NAB(D6) $\rightarrow$ D6 == B5 + C5 + D5	
B6 == B5	
E2 == D2 + D3 $\cdot$ D4	

Table 5.2: Value-based constraint set for the running example from Figure 5.2b.

the MCSes-UnsatisfiableCores algorithm (Algorithm 5.4), both introduced by Liffiton and Sakallah [47], [48]. The later, Algorithm 5.4, is an improved version of Algorithm 5.3. It has increased performance, since it makes use of the SMT solvers' functionality to extract unsatisfiable cores. Otherwise the two algorithms behave the same by enumerating all Minimal Correction Sets (MCSes) of an unsatisfiable formula.

Definition 5.3. Correction set: A correction set is a set of clauses, which needs to be removed from the formula to make it satisfiable.
A correction set *cs* is minimal if there does not exist any subset of *cs*, that is a correction set itself.

This means by finding all MCSes, with either of above mentioned algorithms, we are able to find a set of diagnoses, which tells us which of the spreadsheet's cells are responsible for the test case to fail. Let us have a look at how the algorithms find these MCSes.
Algorithm 5.3 takes as input a formula which is satisfiable, if all not-abnormal variables are unassigned. As output the algorithm provides all minimal diagnoses with increasing cardinality. The cardinality is represented in the algorithm through the variable *upperBound* and is set to one in Line 3 by default. Line 4 performs an initial check, whether the solver is able to find

any diagnoses for the formula. In case the solver returns unsatisfiable the algorithm terminates. Otherwise, the algorithm continues with Line 5, where a weight function is added to a temporary copy of the formula. In the algorithm the weight function is called AtMost and Liffiton and Sakallah defined it like the following [47]. It takes a set of n literals $\{l_1,l_2,\ldots,l_n\}$ and a positive integer k as input and returns a constraint which ensures that:

$$\textsc{AtMost}(l_1, l_2, \ldots, l_n, k) \equiv \sum_{i=1}^{n} \mathbf{value}(l_i) \leq k,$$

where $\mathbf{value}(l_i)$ is one if l_i is assigned true and zero otherwise.

Since SMT solvers do not allow to retrieve all models for a formula at once, the solvers have to be called several times to retrieve all MCSes of maximum cardinality *upperBound*. That is what the inner loop at Line 6 is for. At Line 7 the function GetNewMCS is called. This function tells the solver to find a model and obtains an MCS as follows:

$$\text{MCS} = \{nab \in \text{NAB} : \textit{model(nab)} = \text{true}\}.$$

Since we want the algorithm to find each MCS only once, blocking clauses are added to the original formula φ and to φ' in the Lines 9 and 10. They ensure that the solver returns each MCS only once. A blocking clause for a specific MCS is defined as the disjunction of all not-abnormals included in the MCS.

$$\textsc{BlockingClause}(\text{MCS}) = \bigvee_{nab \in \text{MCS}} nab$$

If all MCSes of the current cardinality are found, meaning $\textsc{solve}(\varphi')$ returns UNSAT, *upperBound* gets incremented by one in Line 12. Then the loop condition from Line 4 is checked again. In case the original formula φ extended by the blocking clauses is still satisfiable, further MCSes are computed. Otherwise, the algorithm terminates and returns the found set of MCSes.

A year after Liffiton and Sakallah introduced the MCS algorithm, they published an improved version, called MCSes-U that makes use of unsatisfiable cores. Most state-of-the-art SMT solvers have the ability to generate a proof of unsatisfiability. As a byproduct an, not necessarily minimal, unsatisfiable core is created, which is a set of variables that led to the for-

Algorithm 5.3 Algorithm to find minimal correction sets [47]

Require: Formula φ with unassigned not-abnormals (NAB)
Ensure: Minimal diagnoses MCSes

```
1:  function MinimalCorrectionsSets(φ)
2:      MCSes := ∅
3:      upperBound := 1
4:      while solve(φ) == SAT do
5:          φ' := φ ∧ AtMost({nab | nab ∈ NAB}, upperBound)
6:          while solve(φ') == SAT do
7:              MCS := GetNewMCS(φ')
8:              MCSes := MCSes ∪ {MCS}
9:              φ' := φ' ∧ BlockingClause(MCS)
10:             φ := φ ∧ BlockingClause(MCS)
11:         end while
12:         upperBound := upperBound + 1
13:     end while
14:     return MCSes
15: end function
```

mula being unsatisfiable. With help of the unsatisfiable core the number of not-abnormals, that need to be considered while calculating MCSes, can be reduced. Meaning all not-abnormals that are not included in the unsatisfiable core set can be assigned to true and therefore, reducing the unassigned variables and shortening the run-time of the algorithm.
The MCSes-U algorithm at first assumes, that all not-abnormals are true, which makes the formula unsatisfiable. Then the GetCore function is executed at Line 4, which calls the SMT solver's function to extract the unsatisfiable core. At Line 5 the SMT solver is called again. This time all not-abnormals are unassigned. If the solver cannot satisfy the formula the algorithm terminates. Otherwise, there has to exist at least one unreported diagnosis. The Instrument function of Line 6 adds the AtMost cardinality to a temporary copy φ'. Additionally, all not-abnormals not included in the unsatisfiable core are set to true, since they are not responsible for the formula being unsatisfiable. Variables included in the core set however, remain unassigned. The loop from Line 8 to 13 produces all MCSes just like Algorithm 5.3. At Line 14 an additional core set is added to the current one. This is done by solving φ' under the assumption that all not-abnormals, which are not included in the current core, are true. The loop of Line 7 checks, if there are any new solutions with the same maximum

cardinality for the changed formula. If this is the case, the inner loop is executed at least one time, otherwise Line 17 is executed, where *upperBound* is incremented by one. When no more solutions can be found the algorithm terminates by returning all computed MCSes.

Algorithm 5.4 Algorithm to find minimal correction sets with unsatisfiable cores [6] (a modified version of [48])

Require: Formula φ with unassigned not-abnormals (NAB)
Ensure: Minimal diagnoses MCSes

```
1:  function MCSesUnsatisfiableCores(φ)
2:      MCSes := ∅
3:      upperBound := 1
4:      core := GetCore(φ, NAB)
5:      while solve(φ) == SAT do
6:          φ' := Instrument(φ, core, upperBound)
7:          while solve(φ') == SAT do
8:              while solve(φ') == SAT do
9:                  MCS := GetNewMCS(φ')
10:                 MCSes := MCSes ∪ {MCS}
11:                 φ' := φ' ∧ BlockingClause(MCS)
12:                 φ := φ ∧ BlockingClause(MCS)
13:             end while
14:             core := core ∪ GetCore(φ', NAB\core)
15:             φ' := Instrument(φ, core, upperBound)
16:         end while
17:         upperBound := upperBound + 1
18:     end while
19:     return MCSes
20: end function
```

5.1.3 Supported Spreadsheet Functions

To debug spreadsheets the framework needs the functionality to convert each spreadsheet function into constraints. There are a lot of different spreadsheet functions and depending on the program the syntax might differ. Therefore, until now the framework only supports a limited amount of these functions. Table 5.3 lists all the supported functions and gives a short description thereof aligned to the equivalent Microsoft Excel functions.

Function	**Description**
Numeric binary operations	addition (+), subtraction (-), multiply ($\cdot$), divide (/) and power ($\hat{}$).
Boolean binary operations	less than ($<$), less or equal ($\leq$), greater than ($>$), greater or equal ($\geq$), equal ($==$) and unequal ($\neq$).
Unary binary operations	unary plus (+) and unary minus (-)
Logical functions	AND ($\wedge$), OR ($\vee$) and NOT ($<>$)
ABS	Returns the absolute value of a supplied number.
AVERAGE	Returns the arithmetic mean of a list of supplied numbers.
IF	Tests a condition and returns one result if the condition is true, and another result if the condition is false.
MAX	Returns the largest value from a list of supplied numbers.
MIN	Returns the smallest value from a list of supplied numbers.
MOD	Returns the remainder from a division between two supplied numbers.
SUM	Returns the sum of a supplied list of numbers.

Table 5.3: List of all spreadsheet functions for which the framework is able to produce Z3 constraints.

5.2 Extensions

As we show in our comparison of different state-of-the-art SMT solvers in Chapter 4, Z3 is currently the only SMT solver suitable for spreadsheet debugging with value-based models. Therefore, we introduce dependency-based models for MBSD of spreadsheets based on the research of Hofer et al. [43]. Furthermore, we introduce a verifying method to improve the quality of dependency-based diagnoses, as well as extend the framework with some additional spreadsheet functions. Figure 5.3 gives an overview of the extended framework focused on the components relevant for SMT solvers. The component shaded in dark gray represents a component with extended functionality. Components which are newly added for this work are shaded in light gray. Note that since Z3 now supports the creation of value-based as well as dependency-based constraint sets, the user needs to specify which approach should be used.

5.2.1 Dependency-based Models

The concept to use dependency-based models for MBSD of spreadsheets is based on the research of Hofer et al. [43]. They state that on the contrary to the value-based models, dependency-based models only propagate the information whether the computed values are correct. Therefore, dependency-based models can be expressed in PL. Meaning we could use any state-of-the-art SAT solver to debug spreadsheets with dependency-based models. However, we can also use SMT solvers, since they all have a SAT solver integrated. Table 5.4 shows that all six compared solvers can be used for MBSD of spreadsheets with dependency-based models. However, only MathSAT 5, SMTInterpol and Z3 can be used in combination with the MCSes-U algorithm.

To create dependency-based constraints we consider each cell as a Boolean. Furthermore, the concrete formulas are ignored and only the cells' dependencies are modeled as a conjunction of the related cells. To do that we make use of implications, instead of equations. This leads to following constraint generation rules for dependency-based models:

$$\forall \text{ cells } c \in Input(\Pi) : c == true$$

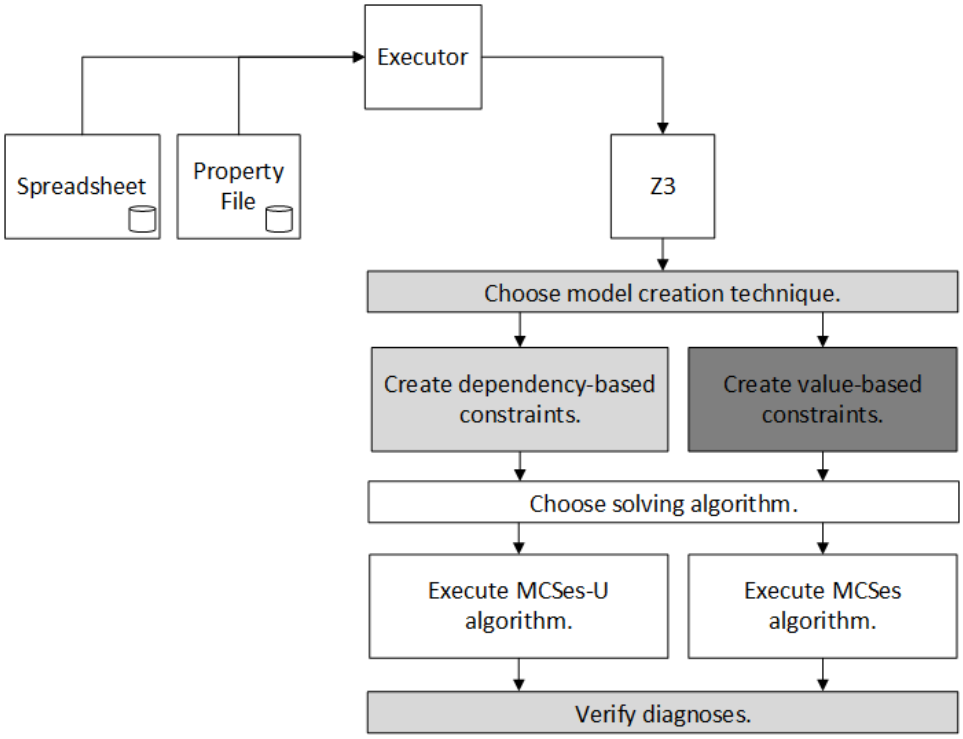


Figure 5.3: Overview of the framework's relevant components for spreadsheet debugging after the expansion. Components shaded in light gray depict the framework's newly added components. The component shaded in dark gray represents a component with extended functionality.

Name	**Decide SAT for PL**	**QF_NA**	**Models**	**Unsat-Cores**
CVC4	yes	yes	yes	no
MathSAT 5	yes	no	yes	yes
SMTInterpol	yes	no	yes	yes
veriT	yes	no	yes	no
Yices 2	yes	no	yes	no
Z3	yes	yes	yes	yes

Table 5.4: Overview of Solvers suitable for dependency-based spreadsheet debugging. Solvers shaded in light gray can be used for dependency-based spreadsheet debugging. Columns shaded in dark gray denote that the solver can be used in combination with the MCSes-U algorithm.

$$\forall \text{ cells } c \in \mathit{Output}(\Pi) \wedge c \in O_{\mathit{correct}} : c == \mathit{true}$$

$$\forall \text{ cells } c \in \mathit{Output}(\Pi) \wedge c \in O_{\mathit{wrong}} : c == \mathit{false}$$

$$\forall \text{ cells } c \in (\bigcup_{c' \in O_{correct}} \textsc{Cone}(c') \setminus \bigcup_{c'' \in O_{wrong}} \textsc{Cone}(c'')) : \bigwedge_{c''' \in \rho(c)} c''' \rightarrow c$$

$$\forall \text{ cells } c \in \bigcup_{c' \in O_{wrong}} \textsc{Cone}(c') : \mathrm{NAB}(\mathrm{cell}_c) \rightarrow \bigwedge_{c'' \in \rho(c)} c'' \rightarrow c$$

with functions $\mathit{Input}(\Pi)$, $\mathit{Output}(\Pi)$, $\textsc{Cone}(c)$ and $\rho(c)$ defined as in Definitions 2.10, 2.11, 2.15 and 2.8 and O_{wrong} and O_{correct} defined as in Definition 2.13. We call this version of a dependency-based model the "simple dependency-based version".

Generating Simple Dependency-based Models

To create simple dependency-based models instead of value-based models we need to make slight adaptions to Algorithm 5.1. The new version of the algorithm, is depicted by Algorithm 5.5.

If we compare Algorithm 5.5 to the one we used to create value-based models, we see that Lines 1 to 10 behave exactly the same. For all cells in O_w we create a conjunct set of the cells' dependencies at Line 12. Line 13 generates the cells' Boolean not-abnormal variables and links them with the cells' simple dependency-based constraints. For the formulas of cells, which are elements of the set O_c, we again create a conjunct set of the cells' dependencies at Line 17. Line 18 adds the cells' simple dependency-based constraints to the auxiliary variable *clause*, which is added to the clause set C_Π at Line 19. At Lines 22 to 27 the constraints for the test case information are generated. Since we consider each variable as a Boolean, we initialize it with either true or false instead of with the cell's value. Finally, the conjunction of the spreadsheet's constraint set C_Π and the test case's constraint set C_T is returned at Line 28.

Algorithm 5.5 Algorithm to convert a spreadsheet into a simple dependency-based model.

Require: Spreadsheet Π, a failing test case T with input I and output O_{wrong} and $O_{correct}$
Ensure: Simple dependency-based clause set of Π and T

1: **function** ConvertSpreadsheetIntoSDBModel(Π, T)

...

11: **for all** cells $c \in O_w$ **do**
12: $D := \bigwedge_{c' \in \rho(c)} \text{Name}(c')$
13: $clause := \text{NAB}(\text{Index}(c)) \rightarrow (D \rightarrow \text{Name}(c))$
14: $C_\Pi := C_\Pi \wedge clause$
15: **end for**
16: **for all** cells $c \in O_c$ **do**
17: $D := \bigwedge_{c' \in \rho(c)} \text{Name}(c')$
18: $clause := D \rightarrow \text{Name}(c)$
19: $C_\Pi := C_\Pi \wedge clause$
20: **end for**
21: $C_T := \emptyset$
22: **for all** cells $c \in I \cup O_{correct}$ **do**
23: $C_T := C_T \wedge (\text{Name}(c) == true)$
24: **end for**
25: **for all** cells $c \in O_{wrong}$ **do**
26: $C_T := C_T \wedge (\text{Name}(c) == false)$
27: **end for**
28: **return** $C_\Pi \wedge C_T$
29: **end function**

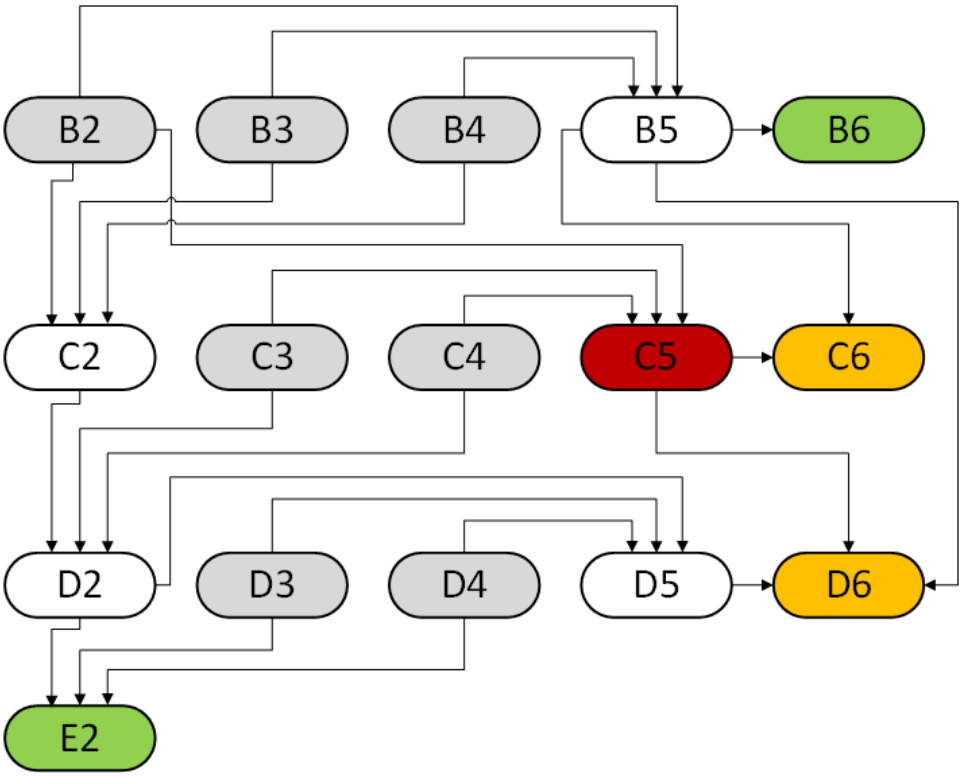


Figure 5.4: Dependency graph of the running example from Figure 5.2b. Gray cells represent the input cells, green cells the correct output cells. The red cell represents the faulty cell and the yellow cells the wrong output cells. An arrow from one Cell X1 to another Y1 denotes the reference of X1 in Y1's formula.

Example 5.2. One important part of creating a simple dependency-based constraint set is to model each cell's dependencies. Therefore, we introduce Figure 5.4 which shows the dependency graph of the running example from Figure 5.2b. An arrow from one Cell X1 to another Y1 denotes the reference of X1 in Y1's formula. Cells with no arrows pointing to them are input cells. Cells with no arrows pointing away from them are output cells. If we apply the simple dependency-based approach to the running example we receive a constraint set as shown in Table 5.5. Solving this constraint set returns two single-fault diagnoses: either Cell B5 or C5 must contain the fault.

The simple dependency-based model leads to more diagnoses, since we use implications to model the cells' relationships with other cells. Therefore, we can only draw conclusions from the input on the output, which leads to

Input Cells:	**Output Cells:**
B2 == true	B6 == true
B3 == true	C6 == false
B4 == true	D6 == false
...	E2 == true
Formula Constraints:	
NAB(B5) $\rightarrow$ B2 $\wedge$ B3 $\wedge$ B4 $\rightarrow$ B5	
NAB(C2) $\rightarrow$ B2 $\wedge$ B3 $\wedge$ B4 $\rightarrow$ C2	
NAB(C5) $\rightarrow$ B2 $\wedge$ C3 $\wedge$ C4 $\rightarrow$ C5	
NAB(C6) $\rightarrow$ B5 $\wedge$ C5 $\rightarrow$ C6	
NAB(D2) $\rightarrow$ C2 $\wedge$ C3 $\wedge$ C4 $\rightarrow$ D2	
NAB(D5) $\rightarrow$ D2 $\wedge$ D3 $\wedge$ D4 $\rightarrow$ D5	
NAB(D6) $\rightarrow$ B5 $\wedge$ C5 $\wedge$ D5 $\rightarrow$ D6	
B5 $\rightarrow$ B6	
D2 $\wedge$ D3 $\wedge$ D4 $\rightarrow$ E2	

Table 5.5: Simple dependency-based constraint set for the running example from Figure 5.2b.

some loss of information. That is why the solution is not as accurate as when using value-based models. However, the problem of information loss can be solved by using "sophisticated dependency-based" models instead.

5.2.2 Sophisticated Dependency-based Model

To get rid of above mentioned weakness of the simple dependency-based model, Hofer et al. [43] suggest to replace the implication with bi-implication (equivalence). By using bi-implication instead of implication conclusions can be drawn from the input on the output and vice versa. However, when using bi-implication we have to be careful of coincidental correctness.

Definition 5.4. Coincidental correctness: Coincidental correctness describes the case where a cell's formula evaluates to the correct value, even though the formula is faulty, or references a faulty cell.

Definition 5.5. Function *ifCC*: The function *ifCC(c)* takes a cell c as input and returns true if coincidental correctness might occur for the formula of c and false otherwise.

Formulas which might lead to coincidental correctness still have to be modeled with implication. Meaning in case of coincidental correctness, similar to the simple dependency-based approach, there occurs some loss of

information. Table 5.6 shows some common spreadsheet functions for which coincidental correctness might occur.

If we change the simple dependency-based constraint generation rules to use bi-implication instead of implication and furthermore, consider coincidental correctness, we get following constraint generation rules for sophisticated dependency-based models:

$$\forall \text{ cells } c \in \Big(\bigcup_{c' \in O_{correct}} \textsc{Cone}(c') \setminus \bigcup_{c'' \in O_{wrong}} \textsc{Cone}(c'') \Big) : \begin{cases} \bigwedge_{c''' \in \rho(c)} c''' \rightarrow c & \textit{ifCC(c)} = \textit{true} \\ \bigwedge_{c''' \in \rho(c)} c''' \leftrightarrow c & \text{otherwise} \end{cases}$$

$$\forall \text{ cells } c \in \Big(\bigcup_{c' \in O_{correct}} \textsc{Cone}(c') \setminus \bigcup_{c'' \in O_{wrong}} \textsc{Cone}(c'') \Big) : \begin{cases} \text{NAB}(\text{cell}_c) \rightarrow \bigwedge_{c'' \in \rho(c)} c'' \rightarrow c & \textit{ifCC(c)} = \textit{true} \\ \text{NAB}(\text{cell}_c) \rightarrow \bigwedge_{c'' \in \rho(c)} c'' \leftrightarrow c & \text{otherwise} \end{cases}$$

with the other rules staying the same. Functions Cone(c), $\rho(c)$ and *ifCC(c)* are defined as in Definitions 2.15, 2.8 and 5.5 and O_{wrong} and $O_{correct}$ are defined as in Definition 2.13.

Generating Sophisticated Dependency-based Models

Since simple dependency-based models and sophisticated dependency-based models look very similar, except for bi-implication instead of implication, we only need to make slight adaptions to Algrotihm 5.5. The new version of the algorithm, is depicted by Algorithm 5.6. However, we need to consider coincidental correctness. That is why we introduce Algorithm 5.7 to check whether coincidental correctness might occur for a certain formula.

If we compare Algorithm 5.6 to the one we used to create simple dependency-based models, the only difference is that we check for coincidental correctness before creating the different constraints (Line 13 and Line 22). In case coincidential correctness might occur the constraints are created the same way as for the simple dependency-based models (Line 14 and Line 23). However, if

Function	Description
∗IF, SUMIF, COUNTIF, ...	In case the condition contains an error the result could still be right. Additionally, if one of the values contains an error but it is not used since the condition is not satisfied the result would still be right. The same is true for all conditional functions.
∗MIN, ∗MAX, ∗COUNT, SMALL, ...	An error in the target area of the function could still lead to a correct result. The same is true for all functions that take an area of cells as input and only pick one of the cells' values as a result.
∗Boolean	$x \vee \mathit{true}$ is always true; $\mathit{false} \wedge x$ is always false; Therefore, no conclusion can be drawn on the value of x.
∗PRODUCT, SUMPRODUCT	The result of a multiplication with 0 is always zero. Therefore, no conclusion can be drawn on the other multiplicand.
∗POWER	No conclusion can be drawn on the exponent if the base equals 0 or 1. Furthermore, if the exponent is 0 no conclusion can be drawn on the base.
MOD	Two entirely different calculations could produce the same result, f.i. $10\%3 = 1$; $31\%3 = 1$.
ROUND, FLOOR, ...	Coincidental correctness can occur for each rounding function.
ABS	The function could produce the right result even if a value with the wrong sign is passed to it.
SIN, COS, ...	For all trigonometric functions which repeat themselves coincidental correctness can occur, f.i. $\cos(0) = 1$; $\cos(2\pi) = 1$.
...	

Table 5.6: Shows some common spreadsheet functions for which coincidental correctness might occur. The occurrence of coincidental correctness for functions preceded by ∗ was shown by Hofer et al. [43]

it is not possible for coincidental correctness to occur we use bi-implication, instead of implication, for the clause creations (Line 16 and Line 25).

Algorithm 5.6 Algorithm to convert a spreadsheet into a sophisticated dependency-based model.

Require: Spreadsheet Π, a failing test case T with input I and output O_{wrong} and $O_{correct}$
Ensure: Sophisticated dependency-based clause set of Π and T

```
1:  function ConvertSpreadsheetIntoSophDBModel(Π, T)
        ...
11:     for all cells c ∈ O_w do
12:         D := ⋀_{c'∈ρ(c)} Name(c')
13:         if IsCoincidentalCorrect(c) then
14:             clause := NAB(Index(c)) → (D → Name(c))
15:         else
16:             clause := NAB(Index(c)) → (D ↔ Name(c))
17:         end if
18:         C_Π := C_Π ∧ clause
19:     end for
20:     for all cells c ∈ O_c do
21:         D := ⋀_{c'∈ρ(c)} Name(c')
22:         if IsCoincidentalCorrect(c) then
23:             clause := D → Name(c)
24:         else
25:             clause := D ↔ Name(c)
26:         end if
27:         C_Π := C_Π ∧ clause
28:     end for
        ...
29: end function
```

Since we need to consider coincidental correctness for sophisticated dependency-based models we introduce Algorithm 5.7. It takes a cell c as input and returns true if coincidental correctness might occur or false otherwise. The algorithm checks all possible cases in which coincidental correctness might occur. In case of nested formulas Lines 2 to 7 check each part of the formula separately for coincidental correctness. If at least one part returns true coincidental correctness occurs. If the formula is not nested we still

Input Cells:	**Output Cells:**
B2 == true	B6 == true
B3 == true	C6 == false
B4 == true	D6 == false
...	E2 == true
Formula Constraints:	
NAB(B5) $\rightarrow$ B2 $\wedge$ B3 $\wedge$ B4 $\rightarrow$ B5	
NAB(C2) $\rightarrow$ B2 $\wedge$ B3 $\wedge$ B4 $\leftrightarrow$ C2	
NAB(C5) $\rightarrow$ B2 $\wedge$ C3 $\wedge$ C4 $\rightarrow$ C5	
NAB(C6) $\rightarrow$ B5 $\wedge$ C5 $\leftrightarrow$ C6	
NAB(D2) $\rightarrow$ C2 $\wedge$ C3 $\wedge$ C4 $\rightarrow$ D2	
NAB(D5) $\rightarrow$ D2 $\wedge$ D3 $\wedge$ D4 $\leftrightarrow$ D5	
NAB(D6) $\rightarrow$ B5 $\wedge$ C5 $\wedge$ D5 $\leftrightarrow$ D6	
B5 $\leftrightarrow$ B6	
D2 $\wedge$ D3 $\wedge$ D4 $\leftrightarrow$ E2	

Table 5.7: Shows the sophisticated dependency-based constraint set for the running example from Figure 5.2b.

need to check all other possibilities in which coincidental correctness can occur. Lines 8 to 11 handle conditional functions and Lines 12 to 16 handle some abstract functions. At Lines 17 to 19 the algorithm checks whether the cell is dependent on a Boolean variable. The case of a multiplication with zero is handled by Lines 20 to 25. Lines 26 to 30 handle the case of a power function with zero or one as a base or zero as an exponent. If no case is found in which coincidental correctness might occur the algorithm returns false at Line 31.

Example 5.3. If the sophisticated dependency-based approach is applied to the running example from Figure 5.2b we receive a constraint set as shown in Table 5.7. Solving this constraint set returns the same single-fault diagnosis as the value-based model: Cell C5 has to be a faulty cell.

5.2.3 Verifying Diagnoses with Value-based models

For simple examples—like our running example—sophisticated dependency-based models provide accurate diagnoses. However, if we add formulas for which coincidental correctness might occur, the sophisticated approach looses its advantage over the simple dependency-based version, since they both model these constraints with implication. Therefore, we introduce a

Algorithm 5.7 Algorithm that decides if coincidental correctness might occur.

Require: Cell c
Ensure: Boolean *isCC* where *isCC* is true when coincidental correctness might occur or false otherwise

```
1:  function IsCoincidentalCorrect(c)
2:      if f(c) is of the form e1 o e2 then
3:          if IsCoincidentalCorrect(e1) ||
4:             IsCoincidentalCorrect(e2) then
5:              return true
6:          end if
7:      end if
8:      if f(c) is of the form IF(cond,e1,e2) || SUMIF(...) ||
9:         COUNTIF(...) then
10:         return true
11:     end if
12:     if f(c) is of the form MIN(...) || MAX(...) || COUNT(...) ||
13:        SMALL(...) || MOD(...) || ROUND(...) || FLOOR(...) ||
14:        ABS(...) || SIN(...) || COS(...) then
15:         return true
16:     end if
17:     if f(c) is a Boolean then
18:         return true
19:     end if
20:     if f(c) is of the form e1 · e2 || PRODUCT(...) ||
21:        SUMPRODUCT(...) then
22:         if ∃ expression e : v(e) == 0 then
23:             return true
24:         end if
25:     end if
26:     if f(c) is of the form POWER(e1,e2) then
27:         if v(e1) == 0 || v(e1) == 1 || v(e2) == 0 then
28:             return true
29:         end if
30:     end if
31:     return false
32: end function
```

method to verify, if dependency-based results are real diagnoses. With this method we can improve the quality of dependency-based diagnoses. However, since we make use of value-based models we can not express the spreadsheet debugging problem in PL anymore. Instead we have to express it in FOL, since we model the cells' formulas and therefore, we are confronted with a non-linear arithmetic problem again. This means that only Z3 can be used to verify dependency-based diagnoses.

To create models for dependency-based diagnosis verification we create a value-based model similar as described in Section 5.1.1. However, we do not make use of not-abnormal variables. Instead we represent each cell—that would otherwise be connected to a not-abnormal variable—equal to its formula's constraint. This leads to a constraint system like the following:

$$\forall \text{ cells } c \in \mathit{Input}(\Pi) : c == v(c)$$

$$\forall \text{ cells } c \in \mathit{Output}(\Pi) : c == v_{exp}(c)$$

$$\forall \text{ cells } c \in \bigcup_{c' \in \mathit{Output}(\Pi)} \textsc{Cone}(c') : c == \mathit{constraint}(f(c))$$

with functions $\mathit{Input}(\Pi)$, $\mathit{Output}(\Pi)$ and $\textsc{Cone}(c)$ defined as in Definitions 2.10, 2.11 and 2.15. To verify if dependency-based diagnoses are valid, we have to try for each of the returned MCSes whether the constraint set is satisfiable if we omit each cell's constraint. In case the constraint set is satisfiable, we mark the MCS as a High Priority Diagnosis (HPD), meaning that it is valid. However, if the constraint set is unsatisfiable, we mark the MCS as a Low Priority Diagnosis (LPD), since it could still be a valid diagnosis in combination with other cells.

Modified Running Example

To illustrate this case we modify our running example from Figure 5.2b. The modified version is shown in Figure 5.5. Figure 5.5a shows the value view of the modified faulty spreadsheet where we manually inserted a second fault in Cell B5—colored in red. We purposefully replaced the acceleration (Cell B3) with the duration (Cell B4) in the calculation of the distance. This can be

	A	B	C	D	E
1		Constant Acceleration	Constant Velocity	Constant Deceleration	Final State
2	Initial Velocity [m/s]	0,0	20,0	20,0	0,0
3	Acceleration [m/s^2]	2,0	0,0	-4,0	
4	Duration [s]	10,0	10.000,0	5,0	
5	Distance [m]	500,0	0,0	50,0	
6	Accumulated Distance [m]	500,0	500,0	550,0	

(a) Modified faulty spreadsheet

	A	B	C	D	E
1		Constant Acceleration	Constant Velocity	Constant Deceleration	Final State
2	Initial Velocity [m/s]	0	=B2+B3*B4	=C2+C3*C4	=D2+D3*D4
3	Acceleration [m/s^2]	2	0	-4	
4	Duration [s]	10	10000	5	
5	Distance [m]	=B2*B4+B4*B4*B4/2	=B2*C4+C3*C4*C4/2	=D2*D4+D3*D4*D4/2	
6	Accumulated Distance [m]	=B5	=B5+C5	=B5+C5+D5	

should be:
B5=B2*B4+B3*B4*B4/2
C5=C2*C4+C3*C4*C4/2

(b) Formula view of the modified faulty spreadsheet

Figure 5.5: Modified running example

seen in the formula view of the modified faulty spreadsheet depicted in Figure 5.5b. Furthermore, it follows that Cell B6 is now an incorrect output cell and therefore, colored in yellow. The test case for the modified example stays the same as for the original example (Table 5.1).

Example 5.4. If we debug the modified faulty spreadsheet from Figure 5.5a with either the simple or the sophisticated approach, we receive the results that Cell B5 is a single-fault diagnosis and (B6, C5) and (B6, C6) are double-fault diagnoses. The value-based approach however, returns only double-fault diagnoses: (B5, B6), (B5, C5), (B5, C6), (B6, C5) and (B6, C6). Therefore, if we verify the solutions from the dependency-based approaches with a value-based model, B5 will be a LPD, since on its own, it is not a valid diagnosis. However, as shown by the result of the value-based approach, B5 is a valid solution in combination with other cells.

Verifying Algorithm

Algorithm 5.8 takes a spreadsheet Π, a failing test case T and a list of dependency-based diagnoses D as input and verifies which diagnoses are valid by splitting them into HPDs and LPDs. First the algorithm creates a constraint set similar to Algorithm 5.1. At Lines 7 to 14 the constraints for the formulas of the output cells are added to the constraint set C_Π. Then the constraints for the test case information are generated and added to the constraint set C_T at the Lines 16 to 21. At Line 22 the two constraint

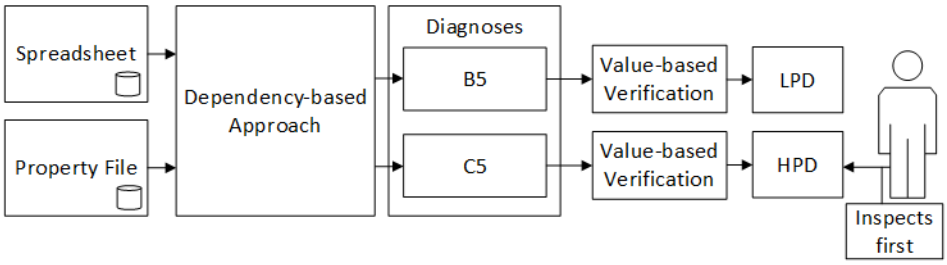


Figure 5.6: Flow chart of the verifying process for the simple approach and running example from Figure 5.2b.

sets C_Π and C_T are added to a combined constraint set CS. Lines 24 to 35 show the verifying process. Each diagnosis mcs contained in D gets verified. Therefore, we omit for each cell contained in mcs the cell's constraint from CS, and store the result in an auxiliary variable CS_{aux} at Lines 26 to 29. Then the solver checks whether CS_{aux} is satisfiable or not (Line 30). In case it is satisfiable the diagnosis is stored in the list of HPDs at Line 31. If CS_{aux} is unsatisfiable the diagnosis is stored in the list of LPDs at Line 33. Finally, at Line 36 the algorithm returns the list HPD as well as LPD.

Example 5.5. Figure 5.6 shows a flow chart of the verifying process for the simple approach and the running example from Figure 5.2b. First a simple dependency-based model is created out of the spreadsheet and the property file. As shown in Example 5.2 the simple approach returns the Cells B5 and C5 as single-fault diagnoses. Both of the diagnoses need to be verified. Table 5.8 and Table 5.9 show the constraint sets for the diagnosis verification. In the constraint set of Table 5.8 Cell B5's constraint is omitted and if passed to the solver, it returns unsatisfiable. In the constraint set of Table 5.9 Cell C5's constraint is omitted and if passed to the solver, it returns satisfiable. This leads to the result that C5 is in fact a valid diagnosis and therefore, is reported as a HPD. However, B5 is only a LPD, meaning it is either no diagnosis at all, or it cannot correct the error of the spreadsheet on its own. Therefore, the user can focus on inspecting the HPDs first.

5.2.4 Extended Spreadsheet Functions

Since the framework only supports a limited number of spreadsheet functions, we add some more common functions to the framework. Table 5.10

Algorithm 5.8 Algorithm to verify dependency-based diagnoses with value-based models.

Require: Spreadsheet Π, a failing test case T with input I and output O, and dependency-based diagnoses D

Ensure: A list of high priority diagnoses *HPD* and a list of low priority diagnoses *LPD*

```
1: function VerifyDiagnoses(Π, T, D)
2:     HPD := ∅
3:     LPD := ∅
4:     CS := ∅
5:     C_Π := ∅
6:     O := ∅
7:     for all cells c ∈ Output(Π) do
8:         O := O ∪ Cone(c)
9:     end for
10:    for all cells c ∈ O do
11:        [C, aux] := ConvertExpressionVB(f(c))
12:        clause := Name(c) == aux
13:        C_Π := C_Π ∧ C ∧ clause
14:    end for
15:    C_T := ∅
16:    for all tuples (c,v) ∈ I do
17:        C_T := C_T ∧ (Name(c) == v)
18:    end for
19:    for all tuples (c,v_exp) ∈ O do
20:        C_T := C_T ∧ (Name(c) == v_exp)
21:    end for
22:    CS := C_Π ∧ C_T
23:    CS_aux := ∅
24:    for all diagnoses mcs ∈ D do
25:        CS_aux := CS
26:        for all cells c ∈ mcs do
27:            [C, aux] := ConvertExpressionVB(f(c))
28:            CS_aux := CS_aux \ {C ∧ aux}
29:        end for
30:        if solve(CS_aux) == SAT then
31:            HPD := HPD ∪ mcs
32:        else
33:            LPD := LPD ∪ mcs
34:        end if
35:    end for
36:    return HPD, LPD
37: end function
```

Input Cells:	**Output Cells:**
B2 == 0	B6 == 100
B3 == 2	C6 == 200,100
B4 == 10	D6 == 200,150
...	E2 == 0
Formula Constraints:	
removed constraint for Cell B5	
C2 == B2 + B3 · B4	
C5 == C2 · C4 + C3 · C4 · C4 / 2	
C6 == B5 + C5	
D2 == C2 + C3 · C4	
D5 == D2 · D4 + D3 · D4 · D4 / 2	
D6 == B5 + C5 + D5	
B6 == B5	
E2 == D2 + D3 · D4	

Table 5.8: Shows the value-based constraint set to verify whether Cell B5 is a valid diagnosis.

Input Cells:	**Output Cells:**
B2 == 0	B6 == 100
B3 == 2	C6 == 200,100
B4 == 10	D6 == 200,150
...	E2 == 0
Formula Constraints:	
B5 == B2 · B4 + B3 · B4 · B4 / 2	
C2 == B2 + B3 · B4	
removed constraint for Cell C5	
C6 == B5 + C5	
D2 == C2 + C3 · C4	
D5 == D2 · D4 + D3 · D4 · D4 / 2	
D6 == B5 + C5 + D5	
B6 == B5	
E2 == D2 + D3 · D4	

Table 5.9: Shows the value-based constraint set to verify whether Cell C5 is a valid diagnosis.

Function	**Description**
PI	Returns the constant value of pi.
POWER	Returns the result of a given number raised to a supplied power.
PRODUCT	Returns the product of a supplied list of numbers.
RANK	Returns the statistical rank of a given value, within a supplied array of values.
SMALL	Returns the k^{th} smallest value from a list of supplied numbers, for a given value k.
SUMPRODUCT	Returns the sum of the products of corresponding values in two or more supplied arrays.
VAR	Returns the variance of a supplied set of values.

Table 5.10: List of newly added spreadsheet functions for which the framework is able to produce Z3 constraints.

lists all the functions we integrate into the framework. Furthermore, we give a short description of each function according to the equivalent Microsoft Excel function.

Chapter 6

Empirical Evaluation

In this chapter we evaluate the value-based and dependency-based approaches in combination with the SMT solver Z3 by means of different spreadsheet corpora. For the evaluation we compare 1) the runtime behavior of the value-based approach with the dependency-based approaches with and without value-based diagnosis verification, 2) the quality of the returned diagnoses of the value-based approach with the dependency-based approaches with and without value-based diagnosis verification, and 3) the distribution of the faulty cells by means of the reported diagnoses. Therefore, in this chapter we describe the spreadsheet corpora used for the evaluation, as well as present the results of the empirical evaluation.

6.1 Spreadsheet Corpus

For the evaluation we make use of a mutated version of the EUSES spreadsheet corpus [37] presented by Hofer et al. [42], which is publicly available at [41]. They filter the EUSES spreadsheet corpus by removing all Excel 5.0 spreadsheets and spreadsheets containing less than five formula cells. For the spreadsheets left, they automatically create up to five mutated spreadsheets by applying mutation operators on randomly chosen formula cells. Following mutation operators are used:

- **Continuous Range Shrinking:** Randomly choosing whether to increment the index of the first column/row, or decrement the index of the last column/row in areas.

- **Reference Replacement:** Randomly changing the row or column index of cell references.
- **Arithmetic Operator Replacement:** Replacing "+" with "-" and vice versa and "·" with "/".
- **Relational Operator Replacement:** Replacing the operators "=", "<", "≤", ">", "≥", and "≠" with one another.
- **Constant Replacement:** Replacing constants with a different constant of the same type.
- **Constant for Reference Replacement:** Replacing cell references with a constant.
- **Formula Replacement with Constant:** Replacing a whole formula with a constant.
- **Formula Function Replacement:** Replacing "SUM" with "AVERAGE", "MIN" with "MAX" and vice versa.

Finally, they check for each mutant whether it is valid, i.e. it does not contain circular references. Furthermore, the inserted fault must be revealed, i.e. at least for one output cell the computed value of the mutant must differ from the value of the original spreadsheet. If either one of these cases does not apply the mutant is discarded. Based on this spreadsheet corpus we remove all spreadsheets containing functions not supported by the framework, leading to a set of 267 spreadsheets for the evaluation. All these spreadsheets contain a single faulty cell. From here on we define these 267 spreadsheets to be the single-fault spreadsheet corpus for the evaluation. These spreadsheets contain both integer and real values. Their content reaches from calculating grades for students, or private issues (e.g. price to remodel a room), to spreadsheets with financial background (e.g. calculations of savings schemes). On average the spreadsheets contain around 105 formula cells, whereas the smallest spreadsheet contains six formula cells and the largest 604 formula cells.

For the last part of the evaluation, we generate double and triple-fault mutants from the EUSES spreadsheet corpus with the same procedure as Hofer

et al. [42]. In total this corpus consists of 217 faulty spreadsheets, whereas 122 spreadsheets contain double-faults and 95 triple-faults. Since these mutants are also based on the EUSES corpus, they have the same attributes as the single-fault spreadsheets. From here on we define these 217 mutants to be the multi-fault spreadsheet corpus.

6.2 Evaluation Results

The evaluation is performed on a computer with an Intel Core i7-3639QM (2,40 GHz quadcore) processor and 8 GB RAM. The framework is running on a 32-Bit Java VM 1.8.0 Update 20 within a 64-Bit Windows 7. For the first two parts of the evaluation we call the framework 25 times to run the value-based approach, as well as the simple and sophisticated approach—later two each with and without value-based diagnosis verification—for each faulty spreadsheet of the single-fault spreadsheet corpus from Section 6.1. Furthermore, we make use of Z3 in combination with the MCSes-U algorithm and set a time limit of five minutes. Finally, we only compute the diagnoses with lowest cardinality, i.e. we only compute double-fault diagnoses, if no single-fault diagnoses are found. Note that from here on we refer to the simple and sophisticated approach with value-based diagnosis verification as the simple and sophisticated approach. If in any case we want to refer to a dependency-based approach without diagnosis verification, we state so explicitly. For the last part of the evaluation we run the value-based approach as well as the simple and sophisticated approach for each faulty spreadsheet of the multi-fault spreadsheet corpus from Section 6.1. Furthermore, we compute all possible diagnoses and not just the ones with the lowest cardinality. The rest of the set-up remains the same. We make use of Z3 in combination with the MCSes-U algorithm and set a time limit of five minutes.

Table 6.1 shows some basic information about the executed runs of the single-fault spreadsheet corpus. For 6.0 % (equals sixteen spreadsheets) of the spreadsheets the value-based approach results in a timeout. The simple and sophisticated approach result in a timeout for 3.0 % (equals eight spreadsheets) of the spreadsheets. Running the dependency-based approaches without value-based diagnosis verification results in both cases in a time-

	Value-based	**Simple**	**Soph.**	**Simple without Verification**	**Soph. without Verification**	**Total**
Timeouts	16	8	8	7	7	16
Timeouts in %	6.0 %	3.0 %	3.0 %	2.6 %	2.6 %	6.0 %
Out of Memory	4	4	3	3	3	5
Out of Memory in %	1.5 %	1.5 %	1.1 %	1.1 %	1.1 %	1.9 %

Table 6.1: The number and percentage of spreadsheets for which each approach results in a timeout or Z3 runs out of memory for the single-fault spreadsheet corpus. Column Total states the number and percentage of **different** spreadsheets for which at least one approach results in a timeout or out of memory.

out for 2.6 % (equals seven spreadsheets) of the spreadsheets. In total there are sixteen different spreadsheets for which at least one of the approaches reaches a timeout. Z3 runs out of memory for 1.5 % (equals four spreadsheets) of the spreadsheets for the value-based and simple approach. For the sophisticated approach, as well as for the dependency-based approaches without diagnosis verification Z3 runs out of memory for 1.1 % (equals three spreadsheets) of the spreadsheets. In total there are five different spreadsheets for which Z3 runs out of memory for at least one approach.
Table 6.2 shows some basic information about the executed runs of the multi-fault spreadsheet corpus. For 3.2 % (equals seven spreadsheets) of the spreadsheets the value-based approach results in a timeout. The simple approach runs out of time for 2.3 % (equals five spreadsheets) of the spreadsheets and the sophisticated approach for 1.8 % (equals four spreadsheets) of the spreadsheets. Running the dependency-based approaches without value-based diagnosis verification results in both cases in a timeout for 1.4 % (equals three spreadsheets) of the spreadsheets. In total there are seven different spreadsheets for which at least one of the approaches reaches a timeout. Finally, Z3 does not run out of memory for any spreadsheet of the multi-fault spreadsheet corpus.

	Value-based	Simple	Soph.	Simple without Verification	Soph. without Verification	Total
Timeouts	7	5	4	3	3	7
Timeouts in %	3.2 %	2.3 %	1.8 %	1.4 %	1.4 %	3.2 %

Table 6.2: The number and percentage of spreadsheets for which each approach results in a timeout for the multi-fault spreadsheet corpus. Column Total states the number and percentage of **different** spreadsheets for which at least one approach results in a timeout.

6.2.1 Runtime Comparison

To calculate each approach's runtime we add up the average solving time over 25 runs for all the spreadsheets of the single-fault spreadsheet corpus. Furthermore, we leave out the runtime of the spreadsheets resulting in a timeout and of those for which Z3 runs out of memory. Table 6.3 shows the runtime behavior of each approach in seconds, as well as the percentage of the required time compared to the fastest approach. The sophisticated approach without diagnosis verification is the fastest with an accumulated average runtime of 44.37 seconds. It is followed by the simple approach without diagnosis verification which on average takes 94.33 seconds. The dependency-based approaches with diagnosis verification take on average 5.3 (sophisticated) and 7.6 (simple) times longer than the fastest approach, and the value-based approach takes on average 7.4 times longer. However, an average runtime for each spreadsheet of 0.18 seconds to 1.37 seconds of the approaches shows that each approach is applicable for real-time spreadsheet debugging. Table 6.4 shows a runtime comparison for each approach with the others. On average the simple approach is for 72.2 % of the spreadsheets faster than the value-based approach and the sophisticated for 62.0 % of the spreadsheets. A comparison of the dependency-based approaches shows that the simple approach is for 59.3 % of the spreadsheets faster than the sophisticated approach. Figures 6.1, 6.2 and 6.3 show the same results as Table 6.4 but in a graphical representation. Based on these results we can infer that spreadsheet debugging with dependency-based models is on average significantly faster than with value-based models. Even with value-based diagnosis verification the dependency-based approaches on average outper-

	Value-based	Simple	Soph.	Simple without Verification	Soph. without Verification
Accumulated Avg. Runtime	329.54 s	337.44 s	234.98 s	94.33 s	44.37 s
Accumulated Avg. Runtime in %	742.7 %	760.5 %	529.6 %	212.6 %	100 %
Average	1.34 s	1.37 s	0.96 s	0.38 s	0.18 s
Median	0.024 s	0.016 s	0.017 s	0.002 s	0.002 s
Stdev	8.33 s	9.29 s	6.17 s	2.65 s	1.08 s

Table 6.3: Accumulated average runtime behavior of each approach over all spreadsheets of the single-fault spreadsheet corpus, with exception to the 21 spreadsheet from Table 6.1 where at least one approach results in either a timeout or out of memory.

	Value-based	Simple	Sophisticated
Value-based	-	72.2 %	62.0 %
Simple	27.8 %	-	40.7 %
Sophisticated	38.0 %	59.3 %	-

Table 6.4: Percentage of spreadsheets for which the "column" approach is faster than the "row" approach.

form the value-based approach. There are however, single cases in which the value-based approach is tremendously faster than the simple approach, leading to the accumulated average runtime of the simple approach to be higher than the value-based approach's. Figure 6.4 and Figure 6.5 show a comparison of the runtime to the number of formula cells and the number of constraints, respectively. They show that there exists a slight correlation between a high runtime and a large number of formula cells and constraints.

6.2.2 Diagnosis Comparison

To evaluate the quality of the diagnoses we make use of the single-fault spreadsheet corpus. Table 6.6 shows the total number of diagnoses returned by each approach and compares them to the value-based approach. The simple approach returns the same amount of diagnoses as the value-based approach. Whereas, the sophisticated approach returns even one diagnosis

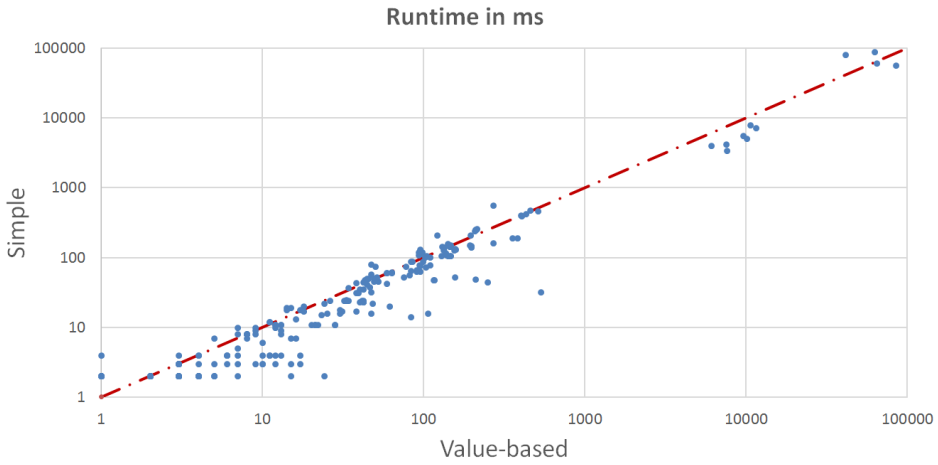


Figure 6.1: Comparison of the average runtime in milliseconds between the value-based and simple approach.

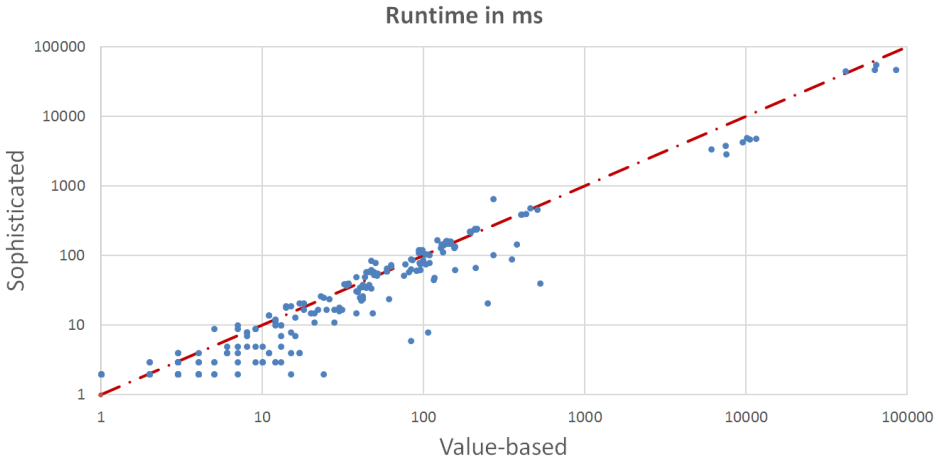


Figure 6.2: Comparison of the average runtime in milliseconds between the value-based and sophisticated approach.

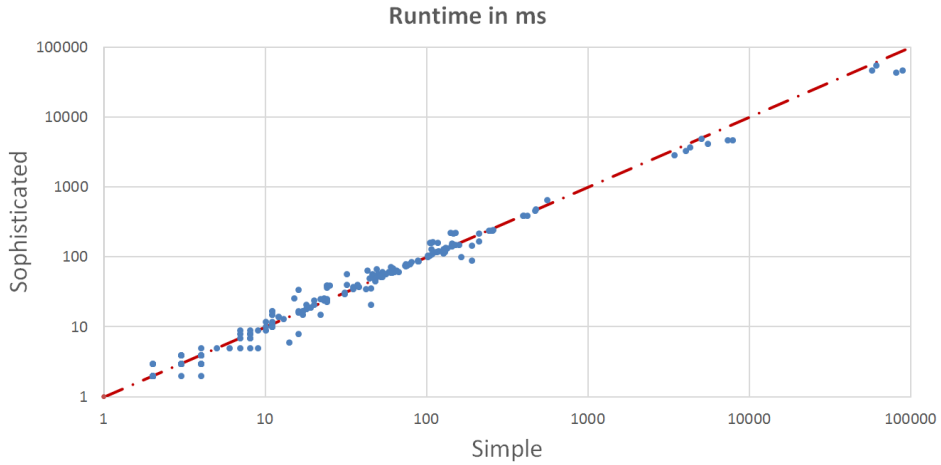


Figure 6.3: Comparison of the average runtime in milliseconds between the simple and sophisticated approach.

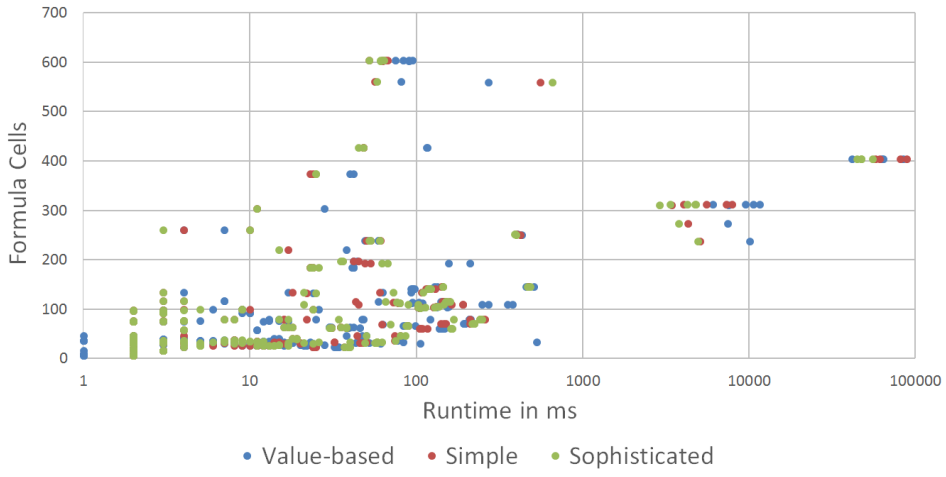


Figure 6.4: Comparison of the runtime with the number of formula cells. There exists a slight correlation between a high runtime and a high number of formula cells.

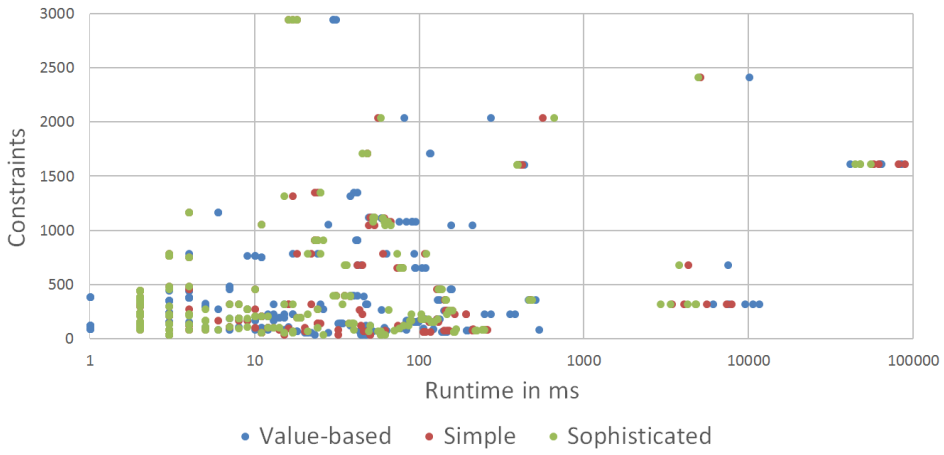


Figure 6.5: Comparison of the runtime with the number of constraints. There exists a slight correlation between a high runtime and a high number of constraints.

less than the value-based approach. This is quite interesting, since it is true for exactly one spreadsheet. The reason therefor is the MCSes-U algorithm which performs an unsatisfiable core extraction on the constraint set before solving it. For the constraint set of the sophisticated approach Z3 is able to reduce the amount of variables that lead to the formula being unsatisfiable to two variables. In case of the simple approach the unsatisfiable core set contains three variables, since the simple approach can model its constraints only with implication, unlike the sophisticated approach which makes use of bi-implication. However, we cannot say for sure why the unsatisfiable core for the value-based constraint set contains also three variables instead of two. The most plausible reason is that Z3 does not necessarily compute a minimal unsatisfiable core. Another reason could be that Z3 makes use of several pre-processing steps to simplify a problem before solving it. This pre-processing steps differ for each theory and therefore, different unsatisfiable cores are produced. Figure 6.6 shows the relevant parts of this special spreadsheet and its formula view. Furthermore, the spreadsheet's faulty test case is depicted in Table 6.5.

The dependency-based approaches without diagnosis verification return 2.42 % or 155 (simple) and 0.20 % or 13 (sophisticated) more diagnoses than the value-based approach. However, if we divide these numbers by the total amount of spreadsheets, we can see that the simple approach without diag-

	A	B	C	D	E	F	G	H	I	J	K	L	M
6				Quiz 1	Quiz 2		Homework 1	Homework 2		Mid-Term		Total Points	Grade
7	Total Points			25	25		20	50		75		195	
8	Last Name	First Name											
9	Bay	Jennifer		23	24		20	46		70		183	94%
10	Carter	Ken		25	23		20	44		66		178	91%
11	Carane	Martha		20	23		20	50		65		178	91%
12	Demcheck	Ike		19	22		20	48		68		177	91%
13	Irwin	Lucy		16	25		20	50		74		185	95%
14	Kriss	James		25	19		20	47		71		182	93%
15	Lutz	Kevin		18	22		20	45		70		175	90%
16	Maines	Larry		11	19		19	45		68		162	83%
17	Neb	Boyde		19	22		20	50		58		169	87%
18	Otter	Mary		20	23		20	48		45		156	80%
19	Thompson	Celeste		25	25		18	49		70		187	34139%
20	Young	Sherry		22	23		20	50		70		185	95%
21	Zare	Tiffany		17	22		20	46		75		180	92%

(a) Special spreadsheet

	A	B	C	D	E	F	G	H	I	J	K	L	M
6				Quiz 1	Quiz 2		Homework 1	Homework 2		Mid-Term		Total Points	Grade
7	Total Points			25	25		20	50		75		=SUMME(D7:K7)	
8	Last Name	First Name											
9	Bay	Jennifer		23	24		20	46		70		=SUMME(D9:K9)	=L9/L$7*1
10	Carter	Ken		25	23		20	44		66		=SUMME(D10:K10)	=L10/L$7*1
11	Carane	Martha		20	23		20	50		65		=SUMME(D11:K11)	=L11/L$7*1
12	Demcheck	Ike		19	22		20	48		68		=SUMME(D12:K12)	=L12/L$7*1
13	Irwin	Lucy		16	25		20	50		74		=SUMME(D13:K13)	=L13/L$7*1
14	Kriss	James		25	19		20	47		71		=SUMME(D14:K14)	=L14/L$7*1
15	Lutz	Kevin		18	22		20	45		70		=SUMME(D15:K15)	=L15/L$7*1
16	Maines	Larry		11	19		19	45		68		=SUMME(D16:K16)	=L16/L$7*1
17	Neb	Boyde		19	22		20	50		58		=SUMME(D17:K17)	=L17/L$7*1
18	Otter	Mary		20	23		20	48		45		=SUMME(D18:K18)	=L18/L$7*1
19	Thompson	Celeste		25	25		18	49		70		=SUMME(D19:K19)	=L19/L$7*356
20	Young	Sherry		22	23		20	50		70		=SUMME(D20:K20)	=L20/L$7*1
21	Zare	Tiffany		17	22		20	46		75		=SUMME(D21:K21)	=L21/L$7*1

(b) Formula view

Figure 6.6: Spreadsheet for which the sophisticated approach produces one diagnosis less than the value-based and simple approach. Cells shaded in gray represent input cells. The correct output cells are colored in green and the faulty cell is colored in red.

Input cells with values:	**Output cells with expected values:**
D7 == 25	M19 == 0.959
D9 == 23	M9 == 0.938
D10 == 25	M10 == 0.913
D11 == 20,000	M11 == 0.913
D12 == 19	M12 == 0.908
D13 == 16	M13 == 0.949
...	...

Table 6.5: A failing test case for the special spreadsheet from Figure 6.6.

nosis verification produces around 0.58 more diagnoses for each spreadsheet than the value-based approach. In case of the sophisticated approach without diagnosis verification the results contain around 0.05 more diagnoses per spreadsheet. This shows that with value-based diagnosis verification it is possible to reduce the number of reported diagnoses for the dependency-based approaches to equal those from the value-based approach. Table 6.7 states the distribution of the reduction. Like Hofer et al. [43] we define the reduction as shown in Definition 6.1.

Definition 6.1. Reduction: Reduction equals the diagnoses' quality and states the percentage of cells which can be excluded from the number of formula cells that need to be manually investigated by a user to fix the faulty spreadsheet.

$$\textsc{Reduction} = 1 - \frac{|\text{Diagnoses in model}|}{|\text{Formula cells}|}$$

The simple and sophisticated approach produce almost the same number of diagnoses as the value-based approach, therefore, they share the same reduction rates. For the simple and sophisticated approach without diagnosis verification we can see that the reduction rates for the other approaches are only slightly better. This leads to the conclusion that with average reduction rates of around 73 %, each of the approaches has the potential to aid users in debugging faulty spreadsheets. Figure 6.7 illustrates the distribution of the reduction rates with respect to the evaluated spreadsheets of the single-fault spreadsheet corpus. It shows that for around 30 spreadsheets not one approach is able to significantly reduce the number of cells that have to be manually investigated (reduction $< 10\,\%$). However, for over 110 spread-

		Value-based	Simple	Soph.	Simple without Verification	Soph. without Verification
	Diagnoses	6400	6400 (HPDs) 155 (LPDs)	6399 (HPDs) 14 (LPDs)	6555	6413
Compared to value-based	**Absolut**	-	0	-1	+155	+13
	In percentage	-	0 %	-0.02 %	+2.42 %	+0.20 %
	Per spreadsheet	-	0	0	+0.58	+0.05

Table 6.6: Compares the diagnoses of the simple and sophisticated approach with and without diagnosis verification to the value-based approach and shows how many more diagnoses per spreadsheet are reported.

sheets all approaches are able to reduce the percentage of formula cells that have to be manually investigated by more than 90 %. Comparing the spreadsheets with low reduction rate to those with high reduction rate results in one major difference. Low reduction rates mostly occur for spreadsheets where many cells are direct data dependent on the faulty cell. Whereas, high reduction rates mostly occur for spreadsheets where few cells are direct data dependent on the faulty cell. Figure 6.8 shows the distribution of the reduction rate with respect to the average runtime in milliseconds for each approach. It can be seen that there exists a correlation between a low reduction rate and a high runtime. As previously stated, low reduction rates mostly occur for spreadsheets where many cells are direct data dependent on the faulty cell, which means that the constraint representations of the spreadsheets contain many not-abnormal variables. The more not-abnormal variables a constraint set contains, the more satisfiability checks have to be performed with the SMT solver which in turn results in a high runtime. On the basis of this information, reasonable termination criteria can be introduced, i.e. the debugging process can be aborted once the runtime exceeds a certain threshold, since most likely no satisfying diagnoses will be found.

Reduction	Value-based	Simple without Verification	Sophisticated without Verification
Average	73.68 %	72.91 %	73.63 %
Median	89.09 %	89.09 %	90.63 %
Stdev	33.27 %	33.27 %	33.28 %

Table 6.7: Distribution of the reduction which equals the diagnoses' quality. The reduction rate of the value-based approach is equal to the reduction rates of the simple and sophisticated approach with diagnosis verification, since they produce almost the same number of diagnoses.

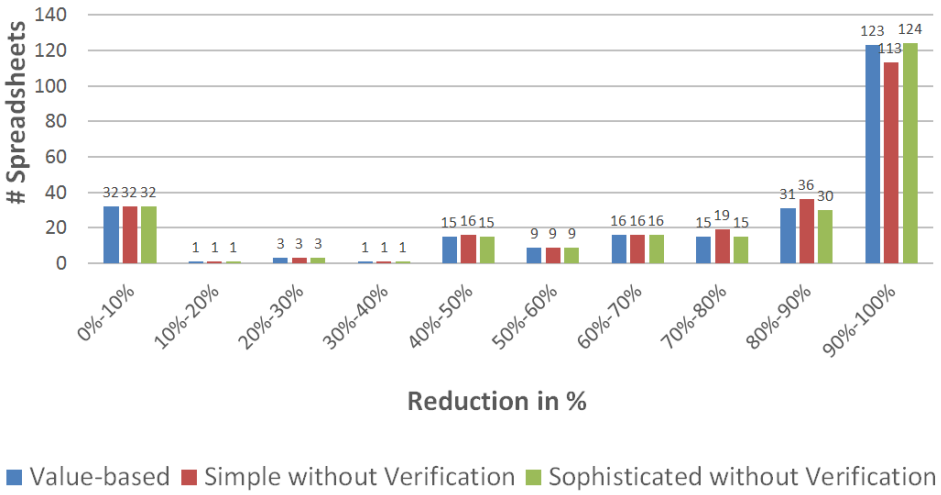


Figure 6.7: Shows an overview of the reduction's distribution for the value-based approach, as well as the simple and sophisticated approach without diagnosis verification.

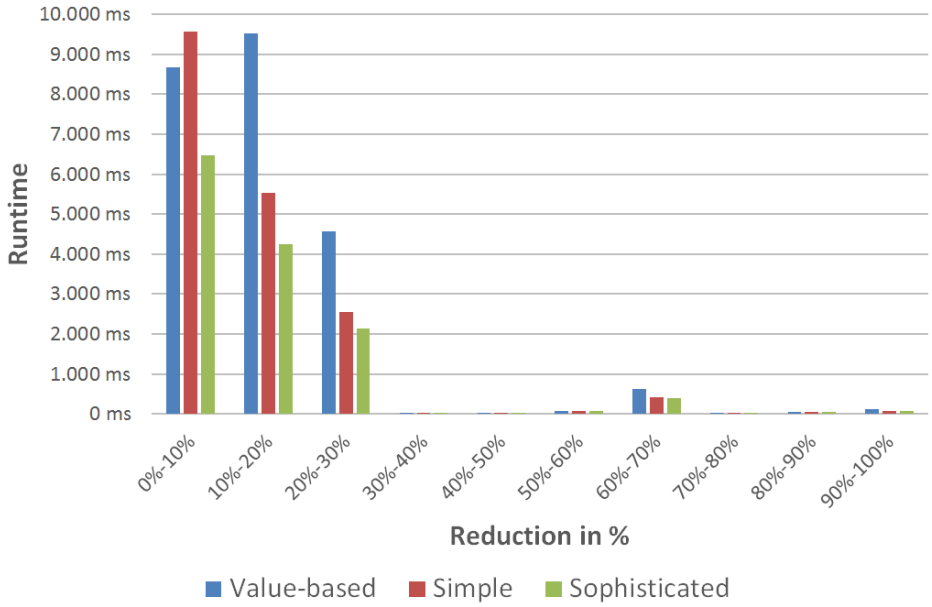


Figure 6.8: Shows an overview of the reduction's correlation to the average runtime for the value-based, simple and sophisticated approach. Low reduction rate equals high runtime. On the basis of this information, reasonable termination criteria can be introduced.

6.2.3 Faulty Cells' Distribution

To evaluate the distribution of the faulty cells we analyze the reported diagnoses for only the multi-fault spreadsheet corpus. For each spreadsheet we split the reported diagnoses into six different categories with decreasing significance: "single-fault HPDs", "single-fault LPDs", "double-fault HPDs", "double-fault LPDs", "triple-fault HPDs" and "triple-fault LPDs". Then we search in each diagnoses category—starting with the most significant one, the single-fault HPDs—for the spreadsheet's faulty cells. Once a faulty cell is found, we report the category it is contained in and continue with the next spreadsheet's diagnoses. Based on the reported categories we can determine the distribution of the faulty cells for each approach which is shown in Figure 6.9. For around 43 % of the spreadsheets all the approaches report the first faulty cell as a single-fault HPD. These high numbers are based on the fact, that even though we only consider multi-faults, the spreadsheet's faulty cells are in many cases connected and therefore, correcting one error could resolve the others. This however, is not always the case, since for around 45 % of the spreadsheets the first found faulty cell is contained in the double-fault HPDs. These are the cases in which the errors are not connected with each other. Meaning correcting one error would not resolve another. Neither of the approaches reported the first found faulty cell in a category with lower significance than the triple-fault HPDs. Furthermore, for only around 0.9 % of the spreadsheets the first found faulty cells are reported as LPDs by both the simple and sophisticated approach. However, in total the sophisticated approach reports fewer LPDs than the simple approach. These are the cases for which the dependency-based approaches find diagnoses with lower cardinality than the value-based approach. However, once these diagnoses get verified by the value-based verification method they will be returned as LPDs. Meaning they can only correct the spreadsheets faults in combination with further cells. As for the value-based approach, no diagnosis verification is needed and therefore, no LPDs are reported. Another interesting fact is that the distributions of the faulty cells for each approach look very alike which shows that the dependency-based approaches with value-based diagnosis verification produce diagnoses of similar quality as the value-based approach.

For an overview of how many cells have to be inspected to first find a faulty cell we give a definition of the best-, average- and worst case, respec-

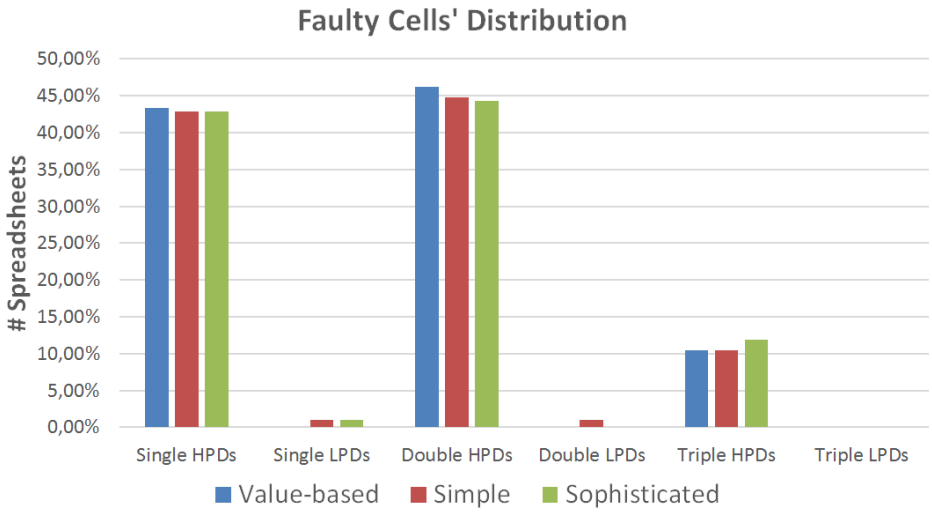


Figure 6.9: Shows the distribution of the faulty cells for each approach based on the results of the multi-fault spreadsheet corpus.

tively. To compute these values the sets of reported diagnoses and their containing cells are not rearranged or changed, but counted from the first reported cell to the last. Since each approach is implemented to first calculate diagnoses with the lowest cardinality—starting with one—and continues with increasing cardinality only if no more solutions can be found, the cells are already in the right order. Therefore, let D be a set containing all reported diagnoses which we split into sets according to their category.

Definition 6.2. Category: Let i represent the category of the reported diagnosis:

$$i = \begin{cases} 1 & \text{single-fault HPDs,} \\ 2 & \text{single-fault LPDs,} \\ 3 & \text{double-fault HPDs,} \\ 4 & \text{double-fault LPDs,} \\ 5 & \text{triple-fault HPDs,} \\ 6 & \text{triple-fault LPDs} \end{cases}$$

Definition 6.3. Diagnoses set: We define D to be the set of all reported diagnoses containing different diagnoses sets of category i, called D^i:

$$D = \bigcup D^i,$$

$$D^i = \{D^i_1, D^i_2, \ldots, D^i_n\},$$

with D^i_j being a diagnosis of D^i containing one or more cells $C^i_{j,k}$:

$$D^i_j = \{C^i_{j,1}, C^i_{j,2}, \ldots, C^i_{j,n}\}.$$

Definition 6.4. Faulty set: Furthermore, we define F to be the set of faulty cells.

$$F = \{C^i_{j,k}, C^i_{j,k}, \ldots\} \qquad \text{with different } C^i_{j,k}$$

Definition 6.5. First faulty cell: C_{Ffirst} represents the first listed faulty cell in D:

$$C_{Ffirst} \in F,$$

$$C_{Ffirst} \in D,$$

$$C_{Ffirst} = \min_k \{\min_j \{\min_i \{C^i_{j,k} \in F\}\}\}.$$

Definition 6.6. Function $\mathbb{1}$: Finally, we define function $\mathbb{1}(C^i_{j,k}, C^{i'}_{j',k'})$ which returns one if $C^i_{j,k}$ is positioned before or at the same position as $C^{i'}_{j',k'}$ in D and zero otherwise.

$$\mathbb{1}(C^i_{j,k}, C^{i'}_{j',k'}) = \begin{cases} 1 & \text{if } (i = i' \wedge j = j' \wedge k \leq k') \vee (i = i' \wedge j < j') \vee (i < i'), \\ 0 & \text{otherwise} \end{cases}$$

With Definitions 6.2 to 6.6 we can further define the best-, average- and worst case for the amount of cells that have to be inspected until the first faulty cell is found in D.

Definition 6.7. Best case: To compute the best case we simply count the number of reported cells, until we find the first faulty cell.

$$\textsc{Best} = \sum_{\forall C^i_{j,k} \in D^i_j \in D} \mathbb{1}(C^i_{j,k}, C_{Ffirst})$$

	Value-based	**Simple**	**Sophisticated**
BEST	10.5	10.8	10.4
AVERAGE	17.3	17.9	17.2
WORST	24.0	25.1	24.0

Table 6.8: The best-, average- and worst cases for the amount of cells that have to be inspected to find the first faulty cell based on the results of the multi-fault spreadsheet corpus.

Definition 6.8. Worst case: The worst case results from the reported diagnoses' total amount of cells.

$$\text{WORST} = \sum_{\forall D_j^i \in D} |D_j^i|$$

Definition 6.9. Average case: The average case represents the arithmetic middle of the best- and worst case.

$$\text{AVERAGE} = \frac{(\text{BEST} + \text{WORST})}{2}$$

Table 6.8 states the best-, average- and worst cases for the amount of cells that have to be inspected to find the first faulty cell within the reported diagnoses for the results of the multi-fault spreadsheet corpus. The best case results in an average amount of around ten cells that need to be inspected. In the average case the number of to inspecting cells rises to around seventeen cells and in the worst case an average of 24 cells from the reported diagnoses need to be considered to find the first faulty cell. Overall, the simple approach is slightly outperformed by the other two approaches. However, as Table 6.9 shows the percentage of more cells that need to be considered in all the cases is minimal. Altogether, the numbers do not spread considerably, proving further that with the value-based diagnosis verification the quality of the dependency-based approaches can be improved to be equal to the quality of the value-based diagnoses.

		Value-based	**Simple**
Compared to sophisticated	Best	+0.41 %	+3.42 %
	Average	+0.17 %	+4.09 %
	Worst	+0.06 %	+4.38 %

Table 6.9: Compares the value-based and simple approach's best-, average- and worst case to the sophisticated approach and shows the percentage of more cells that need to be considered to find the first faulty cell based on the results of the multi-fault spreadsheet corpus.

Chapter 7

Related Work

Supporting programmers with debugging has been an important topic since the computer's early beginnings. Over the years many different automated debugging methods have been introduced. There are many different approaches for automated debugging but mainly there are three different categories: 1) automated fault detection, 2) automated fault localization, and 3) automated fault correction. One approach that can be considered part of all these categories are the source code compilers. They report syntax errors to the programmers, help them to locate errors and sometimes even suggest a solution for fixing them. However, compilers cannot help the programmers with detecting errors in their program's logic. There are however, analytical debugging methods that can. One such method is the automated generation of test cases [39], [66], with the help of test cases, errors can be detected that otherwise would go unnoticed. Furthermore, many automated fault localization and error correction methods are dependent on test cases. Like for example Delta Debugging [22]. Delta Debugging is a fault localization approach that automatically narrows down the set of failure-inducing circumstances to reduce the amount of data the user has to manually inspect for errors. Another automated fault localization method is Program Slicing [69]. Program Slicing reduces faulty programs to a set of statements which are connected to the fault. Yet another automated fault localization approach—upon which our work is based on—is MBSD [58]. It localizes faults by detecting contradictions between the expected- and the obtained output of a program with the help of a model which describes the program's behavior. As for automated fault correction, there are some approaches that

rely on mutation-based repairs [11], meaning modifying the parts related to the fault until it is corrected. Other approaches make use of template-based error correction [46], where they use constraint- or SMT solvers to find a valid initialization for the template's variables and therefore, correct the program. Yet another approach is genetic programming [68]. Genetic programming is inspired by biological evolution. It maintains a population of different programs and mutates them to generate different variants. Then it evaluates these variants by means of a fitness function and selects the suitable variants to correct faults in programs.

MBSD was first introduced by Reiter [58] in 1987. Its concept is applicable for many different areas. Mateis et al. [49], [50] show how MBSD can be utilized to debug Java programs. Friedrich et al. [38] introduce the usage of MBSD for debugging hardware description languages. MBSD can also be applied to debug functional programs as shown by Stumptner et al. [65]. Finally, MBSD can be applied for debugging spreadsheets. Hofer et al. [43] and Abreu et al. [4], [5] make use of MBSD in combination with constraint- and SMT solvers to debug spreadsheets, which builds the basis for our work.

A similar approach is introduced by Jannach and Engler [45], where they make use of an extended Hitting-Set algorithm and user-specified or historical test cases and assertions to solve SDPs. However, their approach slightly differs from ours in some aspects. 1) Jannach and Engler use a constraint solver, whereas we make use of an SMT solver. 2) Instead of using an extended Hitting-Set algorithm, we encode the correct- or incorrectness of the formulas directly into the constraint set. 3) While they require several test cases for their approach to work, our approach relies on only a single test case.

Abraham and Erwig [2] present GoalDebug, a spreadsheet debugger for end users. Whenever the computed output of a cell is incorrect, the user can supply an expected value for the cell. The program then generates a list of change suggestions, which would fix the error if applied. Meaning GoalDebug not only locates errors but also suggests solutions to correct them. The later part is not yet supported by the version presented by Hofer et al. [43] and Abreu et al. [4], [5] and neither by our implementation.

Reichwein et al. [57] developed an incremental testing and debugging tool for spreadsheets. It provides a graphical interface to create and run test cases for spreadsheets. For fault localization it makes use of the dicing approach which is an improved version of program slicing. Additionally to information on incorrect data, dicing makes also use of information on correct data and therefore, further reduces the set of cells the user has to manually inspect.

All of above's approaches, as well as our approach, have one similarity. They only are semi-automatic debugging tools. Meaning, they all are dependent on some kind of user input. Abraham and Erwig [3] introduce UCheck, a fully automated fault localization tool for spreadsheets. It performs a header and unit inference—needing no user input—and reports unit errors.

The WYSIWYT ("What You See Is What You Test") system introduced by Rothermel et al. [61] addresses the topic of spreadsheet testing. Originally, the system helped the user with manual test case creation. A few years later Fisher et al. [36] extended the system with two different automated test case generation techniques. One technique is based on random generation. The other technique is based on a dynamic, goal-oriented approach.

Similar to the WYSIWYT approach Abraham and Erwig [1] present AutoTest, a tool to automatically generate test cases for spreadsheets. It generates test cases by backward propagation and by solving constraints on cell values.

Chapter 8

Conclusion

Considering the vast usage of spreadsheet programs by businesses and private persons it is shocking that there are no common options to automatically debug them. With this work we build upon the research conducted by a team at the Graz University of Technology where they introduce a framework to compare performance and execution time of SMT- and constraint solvers when debugging spreadsheets with value-based MBSD. However, in their work they mainly focus on the constraint solvers. On these grounds we give a general overview of constraint-, SAT- and SMT solvers, how they work and solve specific problems. Furthermore, we conduct a comparison of six different state-of-the-art SMT solvers to find some that can compete with the already integrated SMT solver, Z3, in terms of performance and execution time. To be able to debug spreadsheets, it is important that the solvers can handle real numbers in combination with non-linear arithmetic. Furthermore, the framework's spreadsheet debugging algorithm MCSes-U, which showed the best results in combination with Z3, depends on unsatisfiable core extraction. Therefore, it is equally important that the SMT solvers support that functionality. To our surprise, there exist not many SMT solvers supporting real numbers and from those which do, even less support non-linear arithmetic. In fact there are only six SMT solvers supporting reals that are actively in development, namely CVC4, MathSAT 5, SMTInterpol, veriT, Yices 2 and Z3. From these solvers only Z3 supports non-linear arithmetic. Functionality to extract unsatisfiable cores is provided only by three solvers: MathSAT 5, SMTInterpol and Z3. This leads to the result that currently there exists no solver except Z3, which is a suitable

candidate for spreadsheet debugging with value-based models. Therefore, we extend the framework with two different dependency-based models for MBSD of spreadsheets. On the contrary to value-based models, dependency-based models only propagate the information whether the computed values are correct. Therefore, each cell can be considered as a Boolean, meaning that dependency-based constraint systems can be expressed in PL. With this modification to the framework any state-of-the-art SAT solver, or since most SMT solvers integrate SAT solvers, any of the six compared SMT solvers can be utilized to debug spreadsheets with dependency-based models. However, debugging spreadsheets with the dependency-based approaches may result in less accurate diagnoses than with the value-based approach, due to for example coincidental correctness. To overcome this weakness we introduce a method to verify dependency-based diagnoses with value-based models. Since value-based models take the cells' value into account, we can decide for each dependency-based diagnosis, if it is a real diagnosis or not and divide them into high- and low priority results. Through this process the quality of dependency-based diagnoses can be improved to equal that of the value-based approach. However, since we make use of value-based models we can not express the spreadsheet debugging problem in PL anymore. Instead it is expressed in FOL and therefore, we are confronted with a non-linear arithmetic problem again. Meaning, that value-based diagnosis verification can currently only be conducted with Z3. Furthermore, we compare the dependency-based approaches with and without value-based diagnosis verification to the framework's original value-based approach. Specifically, we answer three questions: 1) which is the fastest approach and are the dependency-based approaches with value-based diagnosis verification faster than the value-based approach, 2) does our method to verify dependency-based diagnoses, improve the quality of the diagnoses enough to equal the quality of the value-based diagnoses, and 3) on average how many cells have to be inspected to find a faulty cell among the reported diagnoses.

The fastest approach is the sophisticated approach without value-based diagnosis verification with an average runtime of 0.18 seconds. The dependency-based approaches with diagnosis verification take on average 5.3 (sophisticated) and 7.6 (simple) times longer than the fastest approach. However, as our research shows they are still considerably faster than the value-based approach. As for the quality of the diagnoses, we show that with the value-

based diagnosis verification method the dependency-based approaches report similar amounts of diagnoses as the value-based approach. However, without value-based verification the results of the dependency-based approaches are slightly worse than the value-based approach's. Furthermore, we show that there is a correlation between a low diagnoses quality and a high runtime. To find a faulty cell within the diagnoses, in the best cases an average of around ten cells need to be inspected. In the average case this number increases to an average of around seventeen cells. Overall, the numbers for each approach are very similar, proving that the value-based diagnosis verification method is effective.

Given these results, the next step in this research is to further improve the runtime of the dependency-based approaches by finding a reasonable termination criteria like aborting the debugging process once the runtime exceeds a certain threshold, since most likely no satisfying diagnoses will be found. Furthermore, due to the extension of dependency-based models it stands to reason to integrate SAT- or SMT solvers into the framework and compare their performance to Z3 when debugging spreadsheets. Finally, and most importantly, after new solvers are integrated in the framework, a number of tests need to be conducted, to evaluate their performance compared to Z3. Such studies will help us assess whether our spreadsheet debugging approaches can be used effectively by end users in the future.

List of Figures

3.1 Example of a 4-queens puzzle 12
3.2 Solution for a map coloring problem 13
3.3 Illustration of backtracking with 4-Queens 15
3.4 Illustration of forward checking with 4-Queens 16
3.5 Overview of the SMT-LIB logics 26
3.6 Congruence closure example 28
3.7 Difference inequalities example 29
3.8 Illustration of the eager and lazy approach 37

5.1 Framework's components for spreadsheet debugging 57
5.2 Running example 63
5.3 Framework's components after the expansion 70
5.4 Dependency graph of the running example 73
5.5 Modified running example 81
5.6 Flow chart of the verifying process 82

6.1 Runtime comparison value-simple 92
6.2 Runtime comparison value-sophisticated 92
6.3 Runtime comparison simple-sophisticated 93
6.4 Comparison of runtime and number of formula cells 93
6.5 Comparison of runtime and number of constraints 94
6.6 Special spreadsheet 95
6.7 Reduction's distribution 98
6.8 Reduction's correlation to runtime 99
6.9 Faulty cells' distribution 101

List of Tables

3.1 SMT-LIB logics' abbreviations 26
3.2 Purification example of the Nelson-Oppen combination method 32

4.1 SMT solvers supporting real numbers 42
4.2 Information of SMT solvers supporting real numbers 43
4.3 SMT solvers' supported theories 44

5.1 Running example's failing test case 63
5.2 Value-based constraint set 64
5.3 Framework's supported spreadsheet functions 68
5.4 SMT solvers for dependency-based spreadsheet debugging . . 70
5.5 Simple dependency-based constraint set 74
5.6 Cases of coincidental correctness 76
5.7 Sophisticated dependency-based constraint set 78
5.8 Value-based constraint set to verify diagnoses 84
5.9 Value-based constraint set to verify diagnoses 84
5.10 Newly added spreadsheet functions 85

6.1 Timeouts and out of memory (single-fault corpus) 89
6.2 Timeouts (multi-fault corpus) 90
6.3 Average runtime 91
6.4 Runtime behavior 91
6.5 Special spreadsheet's failing test case 96
6.6 Overview of diagnoses 97
6.7 Diagnoses' quality (Reduction) 98
6.8 Amount of cells to inspect to find the first faulty cell 103
6.9 Comparison of the to inspecting cells 104

List of Algorithms

5.1 Converting a spreadsheet into a value-based model 60
5.2 Converting an expression into a value-based constraint 62
5.3 Finding minimal correction sets 66
5.4 Finding minimal correction sets with unsatisfiable cores . . . 67
5.5 Converting a spreadsheet into a simple dependency-based model 72
5.6 Converting a spreadsheet into a sophisticated dependency-based model 77
5.7 Deciding if coincidental correctness might occur 79
5.8 Verifying dependency-based diagnoses 83

Acronyms

API Application Programming Interface.

BSD Berkeley Software Distribution.

CDCL Conflict-Driven Clause Learning.

CNF Conjunctive Normal Form.

CP Constraint Programming.

CSP Constraint Satisfaction Problem.

DAG Directed Acyclic Graph.

DPLL Davis-Putnam-Logemann-Loveland.

DPLL(T) Davis-Putnam-Logemann-Loveland modulo Theories.

FOL First-Order Logic.

GPL GNU General Public License.

HPD High Priority Diagnosis.

LFSC Logical Framework with Side Conditions.

LGPL GNU Lesser General Public License.

LPD Low Priority Diagnosis.

MBSD Model-based Software Debugging.

MCS Minimal Correction Set.

MSR-LA Microsoft Research License Agreement.

NP Nondeterministic Polynomial.

PL Propositional Logic.

SAT Boolean Satisfiability.

SDP Spreadsheet Debugging Problem.

SMT Satisfiability Modulo Theories.

SMT-COMP Satisfiability Modulo Theories Competition.

SMT-LIB Satisfiability Modulo Theories Library.

Bibliography

[1] Robin Abraham and Martin Erwig. AutoTest: A tool for automatic test case generation in spreadsheets. In Proceedings of the Visual Languages and Human-Centric Computing, VLHCC'06, pages 43–50, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Robin Abraham and Martin Erwig. GoalDebug: A spreadsheet debugger for end users. In Proceedings of the 29th International Conference on Software Engineering, ICSE'07, pages 251–260, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Robin Abraham and Martin Erwig. UCheck: A spreadsheet type checker for end users. Journal of Visual Languages and Computing, 18(1):71–95, February 2007.

[4] Rui Abreu, Birgit Hofer, Alexandre Perez, and Franz Wotawa. Using constraints to diagnose faulty spreadsheets. Software Quality Journal, pages 1–26, 2014.

[5] Rui Abreu, André Riboira, and Franz Wotawa. Constraint-based debugging of spreadsheets. In Proceedings of the XV Iberoamerican Conference on Software Engineering, Buenos Aires, Argentina, April 24-27, 2012, pages 1–14, 2012.

[6] Simon Ausserlechner, Sandra Fruhmann, Wolfgang Wieser, Birgit Hofer, Raphael Spork, Clemens Muehlbacher, and Franz Wotawa. The right choice matters! SMT solving substantially improves model-based debugging of spreadsheets. In 13th International Conference on Quality Software, pages 139–148, July 2013.

[7] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli.

CVC4. In Proceedings of the 23rd International Conference on Computer Aided Verification, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Clark Barrett, Morgan Deters, Leonardo Mendonça de Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. Journal of Automated Reasoning, 50(3):243–277, 2013.

[9] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). Website. Available at `http://smt-lib.org/`. Visited on July 11th 2014.

[10] Nikolaj Bjørner and Leonardo de Moura. System Description: Z3 0.1, 2007. System Description for the 2007 SMT Competition.

[11] Roderick Bloem, Rolf Drechsler, Görschwin Fey, Alexander Finder, Georg Hofferek, Robert Könighofer, Jaan Raik, Urmas Repinski, and André Sülflow. FoREnSiC - an automatic debugging environment for C programs. In Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, pages 260–265, 2012.

[12] Thomas Bouton, Diego Caminha de Oliveira, David Déharbe, and Pascal Fontaine. veriT: an open, trustable and efficient SMT-solver. In Automated Deduction - CADE-22, volume 5663 of Lecture Notes in Computer Science, pages 151–156. Springer-Verlag, 2009.

[13] Sally Brailsford, Chris Potts, and Barbara Smith. Constraint satisfaction problems: Algorithms and applications. European Journal of Operational Research, 119(3):557–581, 1999.

[14] Polly Brown and John Gould. An experimental study of people creating spreadsheets. ACM Transactions on Information Systems, 5(3):258–272, July 1987.

[15] Roberto Bruttomesso. Satisfiability modulo theories: a pragmatic introduction. Lecture Notes, 2012. Available at `http://www.oprover.org/roberto/teaching/smt/files/manuscript.pdf`. Visited on July 11th 2014.

[16] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. Delayed theory combination vs.

Nelson-Oppen for satisfiability modulo theories: a comparative analysis. Annals of Mathematics and Artificial Intelligence, 55(1-2):63–99, 2009.

[17] Bruno Buchberger. Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal. PhD thesis, University of Innsbruck, 1965.

[18] Cork Constraint Computation Centre. CSP tutorial. Website. Available at `http://4c.ucc.ie/web/outreach/tutorial.html`. Visited on July 11th 2014.

[19] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In Model Checking Software, volume 7385 of Lecture Notes in Computer Science, pages 248–254. Springer Berlin Heidelberg, 2012.

[20] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In Tools and Algorithms for the Construction and Analysis of Systems, volume 7795 of Lecture Notes in Computer Science, pages 93–107. Springer Berlin Heidelberg, 2013.

[21] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient generation of craig interpolants in satisfiability modulo theories. ACM Transactions on Computational Logic, 12(1):7:1–7:54, November 2010.

[22] Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. In 4th International Workshop on Automated Debugging, Munich, Germany, August 2000.

[23] David Cok, David Deharbe, and Tjark Weber. SMT-COMP 2014. Website. Available at `http://smtcomp.sourceforge.net/2014/`. Visited on July 11th 2014.

[24] David Cok, Alberto Griggio, Roberto Bruttomesso, and Morgan Deters. The 2012 SMT Competition. In SMT 2012, volume 20 of EPiC Series, pages 131–142, 2012.

[25] Jeremy Condit and Matthew Harren. Congruence closure. Lecture Notes. Available at `http://www.cs.berkeley.edu/~necula/autded/lecture12-congclos.pdf`. Visited on July 11th 2014.

[26] Jácome Cunha, João Paulo Fernandes, Hugo Ribeiro, and João Saraiva. Towards a catalog of spreadsheet smells. In 12th International Conference Computational Science and Its Applications, pages 202–216, 2012.

[27] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. Communications of the ACM, 5(7):394–397, July 1962.

[28] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. Journal of the ACM, 7(3):201–215, July 1960.

[29] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. Electronic Notes in Theoretical Computer Science, 198(2):37–49, May 2008.

[30] Leonardo de Moura and Nikolaj Bjørner. Proofs and refutations, and Z3. In Logic for Programming Artificial Intelligence and Reasoning Workshops, volume 418 of CEUR Workshop Proceedings, 2008.

[31] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08 / ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] David Déharbe, Pablo Federico Dobal, and Pascal Fontaine. veriT: System description for SMT-COMP 2014, 2014. System Description for the 2014 SMT Competition.

[33] Bruno Dutertre. Yices 2 Manual. SRI International, Menlo Park, CA. Available at `http://yices.csl.sri.com/manual.pdf`. Visited on July 11th 2014.

[34] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In Proceedings of the 18th International Conference on

Computer Aided Verification, CAV'06, pages 81–94, Berlin, Heidelberg, 2006. Springer-Verlag.

[35] Alessandro Farinelli. Propositional and first order logic. Lecture Notes. Available at `http://profs.sci.univr.it/~farinelli/courses/ar/slides/prop-fol.pdf`. Visited on July 11th 2014.

[36] Marc Fisher, Mingming Cao, Gregg Rothermel, Curtis Cook, and Margaret Burnett. Automated test case generation for spreadsheets. In Proceedings of the 24th International Conference on Software Engineering, ICSE'02, pages 141–153, New York, NY, USA, 2002. ACM.

[37] Marc Fisher and Gregg Rothermel. The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In Proceedings of the First Workshop on End-user Software Engineering, Workshop on End-User Software Engineering I, pages 1–5, New York, NY, USA, 2005. ACM.

[38] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. Artificial Intelligence, 111(1-2):3–39, July 1999.

[39] Peter Fröhlich and Johannes Link. Automated test case generation from dynamic models. In European Conference on Object-Oriented Programming 2000 (ECOOP'00), volume 1850 of Lecture Notes in Computer Science, pages 472–491. Springer Berlin Heidelberg, 2000.

[40] The ACSys Group. CVC4 User Manual. New York University and University of Iowa. Available at `http://cvc4.cs.nyu.edu/wiki/User_Manual`. Visited on July 11th 2014.

[41] Birgit Hofer, André Riboira, Franz Wotawa, Rui Abreu, and Elisabeth Getzner. Mutated EUSES corpus. Website. Available at `https://dl.dropboxusercontent.com/u/38372651/Spreadsheets/EUSES_Spreadsheets.zip`. Visited on July 11th 2014.

[42] Birgit Hofer, André Riboira, Franz Wotawa, Rui Abreu, and Elisabeth Getzner. On the empirical evaluation of fault localization techniques for spreadsheets. In Proceedings of the 16th International Conference

on Fundamental Approaches to Software Engineering, FASE'13, pages 68–82, Berlin, Heidelberg, 2013. Springer-Verlag.

[43] Birgit Hofer and Franz Wotawa. Why does my spreadsheet compute wrong values? In IEEE International Symposium on Software Reliability Engineering, pages 112–121, 2014.

[44] Adele Howe. Constraint satisfaction problems. Lecture Notes. Available at `https://www.cs.colostate.edu/~howe/cs440/csroo/F2013/more_progress/09_csp2013.pdf`. Visited on July 11th 2014.

[45] Dietmar Jannach and Ulrich Engler. Toward model-based debugging of spreadsheet programs. In 9th Joint Conference on Knowledge-Based Software Engineering, pages 252–264, August 2010.

[46] Robert Könighofer and Roderick Paul Bloem. Automated error localization and correction for imperative programs. In IEEE, editor, Proceedings of 11th International Conference for Formal Methods in Computer Aided Design, pages 91 – 100. IEEE, 2011.

[47] Mark Liffiton and Karem Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning, 40(1):1–33, January 2008.

[48] Mark H. Liffiton and Karem A. Sakallah. Generalizing core-guided Max-SAT. In Theory and Applications of Satisfiability Testing - SAT 2009, volume 5584 of Lecture Notes in Computer Science, pages 481–494. Springer Berlin Heidelberg, 2009.

[49] Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. Model-based debugging of Java programs. In Automated analysis-driven debugging, 2000.

[50] Cristinel Mateis, Markus Stumptner, Franz Wotawa, and Technische Universität Wien. Locating bugs in Java programs – first results of the Java diagnosis experiments (jade) project. In In Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, 2000.

[51] Wolfgang Mayer and Markus Stumptner. Model-based debugging using multiple abstract models. Computing Research Repository, 2003.

[52] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In Formal Methods: Foundations and Applications, pages 23–36. Springer-Verlag, Berlin, Heidelberg, 2009.

[53] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. Communications of the ACM, 54(9):69–77, September 2011.

[54] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM, 53(6):937–977, November 2006.

[55] Albert Oliveras and Enric Rodriguez-Carbonell. Combining decision procedures: The Nelson-Oppen approach. Lecture Notes. Available at `http://www.lsi.upc.edu/~oliveras/TDV/NO.pdf`. Visited on July 11th 2014.

[56] Ray Panko. SSR-Spreadsheet Research. Website. Available at `http://panko.shidler.hawaii.edu/SSR/`. Visited on July 11th 2014.

[57] James Reichwein, Gregg Rothermel, and Margaret Burnett. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. J-SIGPLAN, 35(1):25–38, January 2000.

[58] Raymond Reiter. A theory of diagnosis from first principles. Artificial Intelligence, 32(1):57–95, April 1987.

[59] Microsoft Research. Z3 Website. Website. Available at `http://z3.codeplex.com/`. Visited on July 11th 2014.

[60] Andrew Reynolds, Liana Hadarean, Cesare Tinelli, Yeting Ge, Aaron Stump, and Clark Barrett. Comparing proof systems for linear real arithmetic with LFSC. In Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, Scotland, July 2010.

[61] Karen Rothermel, Curtis Cook, Margaret Burnett, Justin Schonfeld, Thomas Green, and Gregg Rothermel. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In Proceedings of the

22nd International Conference on Software Engineering, ICSE'00, pages 230–239, New York, NY, USA, 2000. ACM.

[62] Karem Sakallah and Igor Markov. ACM SIGDA newsletter, volume 36, number 24. Newsletter, December 2006. Available at `http://archive.sigda.org/newsletter/2006/eNews_061215.html`. Visited on July 11th 2014.

[63] Helmut Simonis. Building industrial applications with constraint programming. In Constraints in Computational Logics, volume 2002 of Lecture Notes in Computer Science, pages 271–309. Springer Berlin Heidelberg, 2001.

[64] Aaron Stump, Tjark Weber, and David Cok. Progress report on the 2013 SMT evaluation. Presentation Slides. Available at `http://sat2013.cs.helsinki.fi/slides/SMTEVAL2013.pdf`. Visited on July 11th 2014.

[65] Markus Stumptner and Franz Wotawa. Debugging functional programs. In Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2, IJCAI'99, pages 1074–1079, 1999.

[66] Nigel Tracey, John Clark, and Keith Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. International Workshop on Dependable Computing and Its Applications, 1998.

[67] Edward Tsang. Foundations of constraint satisfaction. Computation in cognitive science. Academic Press, 1993.

[68] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In Proceedings of the 31st International Conference on Software Engineering, ICSE'09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[69] Mark Weiser. Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1979.

[70] Lintao Zhang. SAT-Solving: From Davis-Putnam to Zchaff and Beyond. Presentation Slides. Available at `http://www.cfdvs.iitb.ac.in/download/Docs/verification/papers/sat/reviews-and-tutorials/sat_course1.pdf`. Visited on July 11th 2014.