716.117 Diploma Seminar

SMT Solver Comparison

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, July 2014

Abstract

Over the past years Satisfiability Modulo Theories (SMT) solvers have become a very popular tool to solve different kinds of problems. Due to a high level operation language and the usage of state-of-the-art Boolean Satisfiability (SAT) solvers, SMT solvers can not only solve SAT problems but even more complex ones. Recent research introduced the usage of SMT solvers for model-based debugging of spreadsheets. A framework was developed to compare the performance and runtime of different modern solvers when debugging spreadsheets. This paper builds upon this research by giving a brief description of SMT solvers and by comparing different notable solvers, regarding their functionality, only considering solvers able to handle real numbers. Surprisingly not many SMT solvers fall into that category. That is why, within the scope of this work, we pick out the suitable SMT solvers and suggest an efficient method to integrate them into the framework.

Contents

Chapter 1

Introduction

Spreadsheet programs, like Microsoft's Excel, OpenOffice's Calc or Apple's Numbers, are some of 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 tools 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. Even though spreadsheets are so popular, it is very difficult to automatically debug them. There are many different approaches that try to handle spreadsheet debugging, yet, they are 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 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 in comparison to constraint solvers when debugging spreadsheets, we like to build on some recent research done by a team at the Graz University of Technology. They developed a framework, which compares different SMT- and constraint solvers based on their execution time and performance when debugging spreadsheets [1]. 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, 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 was mainly focused on integrating the constraint solvers and it remains unclear, whether other SMT solvers would yield similar performance as Z3 when debugging spreadsheets. That is why, next to providing a general introduction to the basics of SMT solvers, we will offer a comparison of different state-of-the-art SMT solvers, regarding their functionality, only considering solvers that are able to operate with real numbers. Furthermore, a translation of a spreadsheet into a spreadsheet debugging problem results in a non-linear arithmetic problem. That is why, 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 from those which do, even less support non-linear arithmetic and unsatisfiable core extraction. That is why within the scope of this work we pick out the suitable SMT solvers and introduce an efficient way to integrate them into the framework, to ease the way for a future work where we will integrate them to compare their performance and execution time when debugging spreadsheets.

The following pages of this paper are organized as follows: Chapter 2 deals with the principles of SMT solvers. We give a detailed description of their input language, the problems they are designed to solve and how they operate to solve these problems. Chapter 3 consists of a comparison of state-of-the-art SMT solvers that support real numbers. We give a short summary of their basic functionality as well as a detailed description of their technical features. In Chapter 4 we state two different modeling languages for SMT solvers and show with the aid of an example how their syntax differs from each other. Finally, we conclude the paper in Chapter 5.

Chapter 2

Satisfiability Modulo Theories 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 [30], [36].
Within this chapter we describe the basic technology of SMT solvers and the kind of problems they are designed to solve.

2.1 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 [24]. Definitions 2.1 to 2.9, are based on Farinelli's description of FOL [24].

Definition 2.1. Variables and Constants: The language of FOL consists of variables with values of various types (f.i. Boolean, integer, real), as well as constants;

Definition 2.2. Operators: Additionally, to the operators of Propositional Logic (PL), negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), FOL makes use of the equals operator ($=$);

Definition 2.3. 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 2.4. 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 2.1. Assuming, *Person* is a predicate symbol with arity 1, then $Person(x)$ would evaluate to *true* only if x really is a person.

Example 2.2. 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*, otherwise *false*.

Definition 2.5. 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 2.3. 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 2.6. Terms: Every variable and constant on its own is a term. Furthermore, if f is a function with arity n and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term as well.

Definition 2.7. Formulas: Every Boolean variable is a formula. Furthermore, if P is a predicate with arity n and $t_1, ..., t_n$ are terms, then $P(t_1, ..., t_n)$ is a formula as well. Terms linked by any operator are also formulas.

Definition 2.8. Sentences: A formula with no free variable is called a sentence.

Definition 2.9. Atomic formulas: A formula containing no logical connective and no bound variable is called an atomic formula or an atom.

2.2 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. Typical examples of such theories are 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 or 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 Nondeterministic Polynomial (NP) complete. Furthermore, FOL is undecidable and the computational complexity of most SMT problems is very high. That is why most solvers focus on efficiently solving practical problems, like formulas produced by verification and analysis tools, since most of these formulas can be efficiently solved. SMT additionally struggles with finding algorithms that efficiently handle different theories and also work well when 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 [29].

2.3 Theories

One core part of SMT is made up of 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

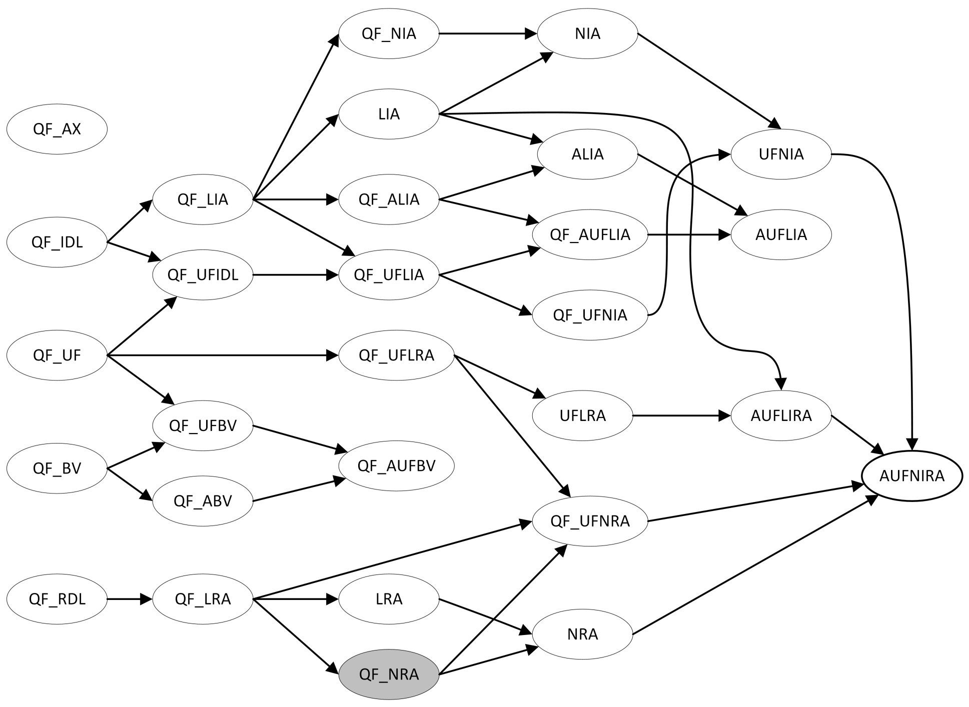


Figure 2.1: Overview of the SMT-LIB logics [4]. A link from a logic L1 to a logic L2 means that every formula of L1 is also a formula of L2. The logic shaded in gray is the one relevant for spreadsheet debugging.

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) [4] 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 2.1 shows an overview of the SMT-LIB logics. The abbreviations' meanings of these logics are explained in Table 2.1.

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 2.1: Short explanation of SMT-LIB logics' abbreviations [4].

2.3.1 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 [29].

2.3.2 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 [15], 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 [15]:

Definition 2.10. Reflexivity: $\frac{}{E=E}$

Definition 2.11. Transitivity: $\frac{E_1=E_2 \quad E_2=E_3}{E_1=E_3}$

Definition 2.12. Symmetry: $\frac{E_2=E_1}{E_1=E_2}$

Definition 2.13. Congruence: $\frac{E_1=E_2}{f(E_1)=f(E_2)}$

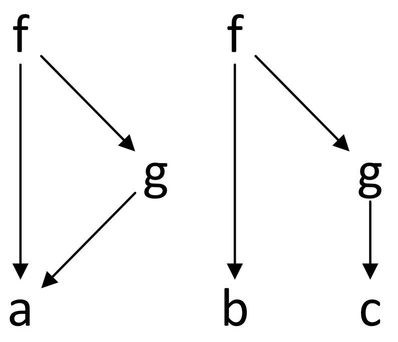
Decision procedures for this theory have great significance, since the decision problem for other theories can be reduced to it. Many theory solvers for uninterpreted functions are based on the *congruence closure* method. If we consider a formula, which consists of conjunctions of equalities between 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 [29]. Figure 2.2 shows a step-by-step example of the congruence closure algorithm.

2.3.3 Linear arithmetic

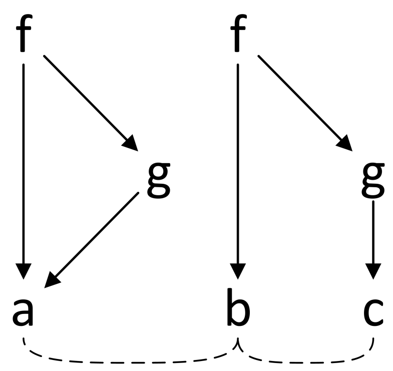
The theory of linear arithmetic is denoted by the SMT-LIB as LIA, LRA, QF_LIA and QF_LRA, which stands for quantified or quantifier-free linear integer arithmetic or linear real arithmetic. Their definitions state that arithmetical functions +, - and $\cdot$ are supported. However, $\cdot$ is restricted 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 $:$ is also allowed: $c : x$, where c is a rational coefficient and x a variable. Furthermore, relational symbols for equality and inequalities $(=, \leq, <, \ldots)$ are used to form atomic predicates. A popular procedure for deciding linear arithmetic, which many SMT solvers use in its linear arithmetic solver, is called the *simplex* algorithm [29]. Dutertre and de Moura explained in detail how this algorithm works and presented a more efficient version for linear arithmetic solvers for DPLL(T) in [23].

2.3.4 Difference arithmetic

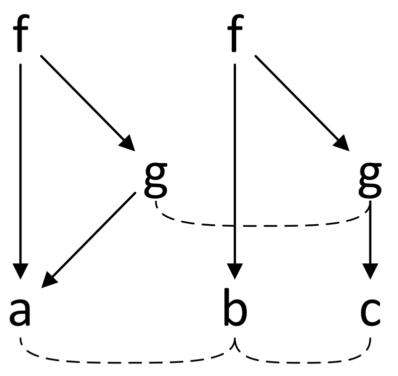
The theory of difference arithmetic is denoted by the SMT-LIB as QF_IDL and QF_RDL, which stands for quantifier-free integer difference logic and quantifier-free real difference logic. It is a part of linear arithmetic, where inequalities are restricted to have the form $x - y \leq c$, for variables x, y and



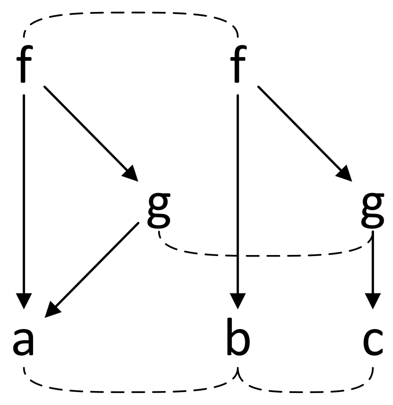
(a) DAGs, each representing a term of the example;



(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);

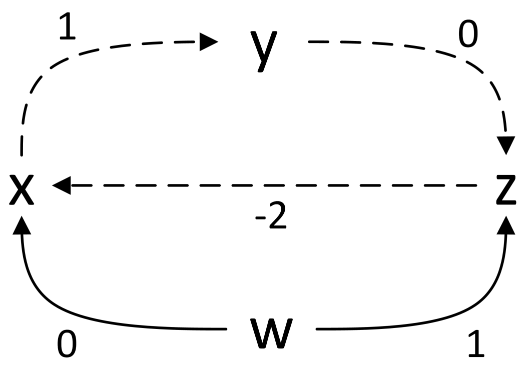


(d) Nodes $f(a, g(a))$ and $f(b, g(c))$ are also congruent, since $a = c$ and $g(a) = g(c)$. Therefore, the example is unsatisfiable because the term $f(a, g(a)) \neq f(b, g(c))$ is not true;

Figure 2.2: Congruence closure example: $a = b \wedge b = c \wedge f(a, g(a)) \neq f(b, g(c))$ [29].

$$\begin{aligned} y - x &\leq 1 \\ z - y &\leq 0 \\ x - z &\leq -2 \\ x - w &\leq 0 \\ z - w &\leq 1 \end{aligned}$$

(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 2.3: Difference inequalities example [29].

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 [29]. Figure 2.3 shows an example of a conjunction of difference inequalities, as well as its representation as a graph.

2.3.5 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.* 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 [29]. Adding quantifiers to the theory makes it even worse. According to [29], 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 be able to debug spreadsheets.

2.3.6 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* [29].

2.3.7 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 2.14. **write**(a,i,v): writes value v at index i of array a.

Definition 2.15. **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. 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* [29].

2.3.8 Quantified Theories

If we consider quantifiers part of the language of FOL the problem of deciding satisfiability becomes significantly more difficult. In fact not many SMT solvers support theories that allow quantifiers. However, if quantifiers are supported, usually some form of *E-matching* is performed to decide satisfiability.

2.3.9 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

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 2.2: Purification example of the Nelson-Oppen combination method.

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 [29], [8]:** 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 [32]. 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:

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

  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 2.2.

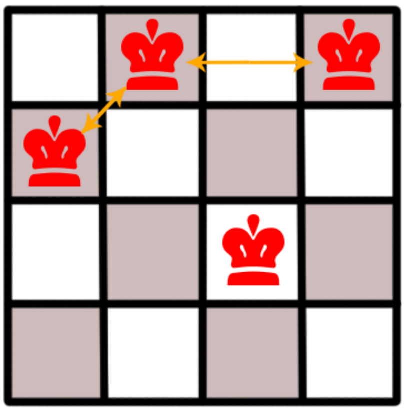
  With that the two theory solvers can check satisfiability for their part of the formula, while propagating entailed equalities of their shared variables between them. This is done until a convergence is reached, meaning, the formula is satisfiable, or until one solver returns unsatisfiable.

- **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 [8].

- **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 [18].

2.4 Famous Problems expressed as SMT

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


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

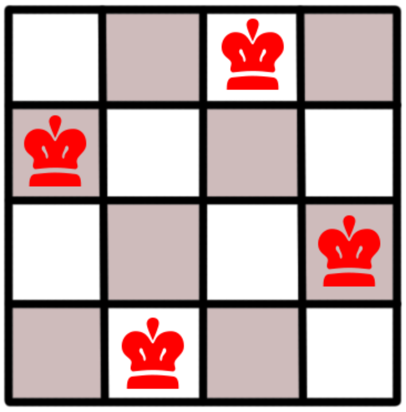

(b) A valid solution for the 4-queens puzzle;

Figure 2.4: Example of a 4-queens puzzle.

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 an SMT problem can look as follows:

- **Variables:** $p_{1,1}, p_{1,2}, p_{1,3}, p_{1,4}, p_{2,1}, \ldots, p_{4,3}, p_{4,4}$ (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

  $(p_{1,1} + p_{1,2} + p_{1,3} + p_{1,4} \ = \ 1) \wedge \ldots$ (for each row and column) $\wedge$
  $(p_{1,1} + p_{2,2} + p_{3,3} + p_{4,4} \leq 1) \wedge \ldots$ (for each diagonal)

Figure 2.4 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.

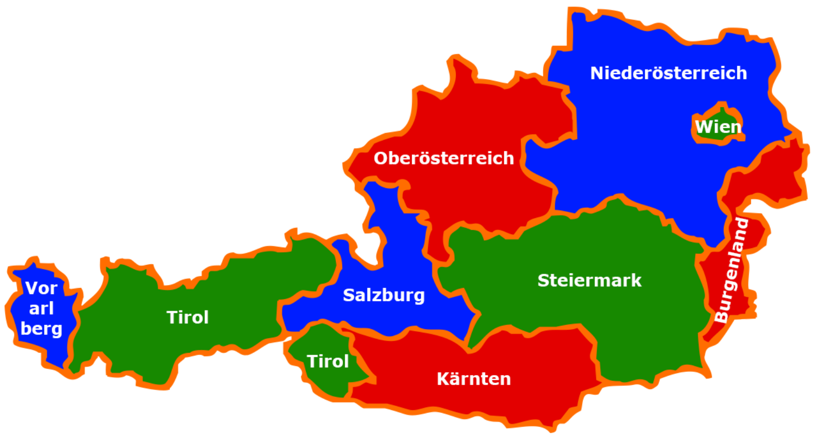


Figure 2.5: A possible solution of the map coloring problem for the map of Austria.

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

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

Figure 2.5 shows a possible solution of the map coloring problem for a map of Austria.

2.5 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 was 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 Conjunctive Normal Form (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 will lead 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 [7]. 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 put into creating algorithms like that, the **lazy** approach is in many cases tremendously faster [31].

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 a FOL formula [7]:

- 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 2.4. $x = y \land (x < y \lor x > y)$
According to above 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 $\neg(a \land b)$ and $\neg(a \land c)$. We now translate the FOL formula into a PL one by converting every atom into a Boolean variable and by adding all found inconsistencies. This will lead to the result $a \land (b \lor c) \land \neg(a \land b) \land \neg(a \land c)$. 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 value.

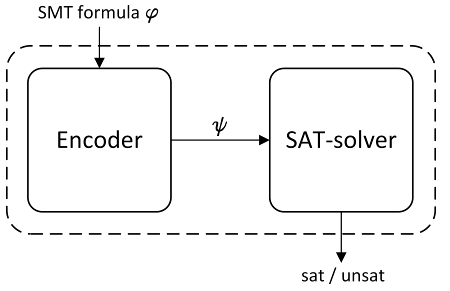
Lazy approach

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 than as with the **eager** approach. Nonetheless, the **lazy** approach is, due to its flexibility, the more commonly used procedure in existing SMT solvers [7].

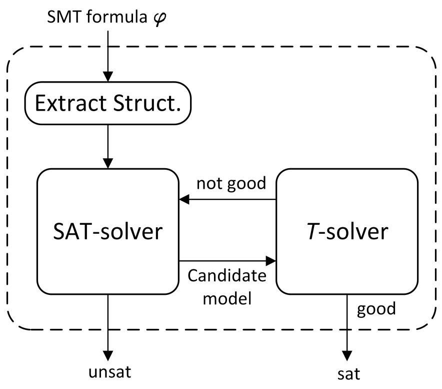
Following solving methodology gives a general idea on how an SMT solver, that interacts with its SAT solver according to the lazy approach, decides satisfiability for a FOL formula [7]:

- 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 SAT solver and begin anew with deciding its satisfiability;

Example 2.5. $x = y \wedge (x < y \vee x > y)$
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



(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 2.6: Illustration of the eager and lazy approach [7].

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 will add this inconsistency, $\neg(a \wedge b)$, to the formula and pass 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 2.6 shows a high-level view of both the eager and lazy approach.

2.6 DPLL and DPLL(T)

Most modern SAT solver are based on the Davis-Putnam-Logemann-Loveland (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. Following sections introduce an abstract DPLL model, as well as an abstract DPLL(T) model.

2.6.1 DPLL paradigm

The 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 [17], [16]. It consists of the following seven transition rules, which describe in a general way how modern DPLL based SAT solvers work.

UnitPropagate

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

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 was not yet defined by M. The only way for the clause to be true in M is to extend M with the remaining literal equal to true [31].

Decide

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

Decide conducts a case split. It chooses an undefined literal l and 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 [31].

Fail

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

If a *conflicting clause* gets detected and M contains no decision literals, meaning, all literals in M have a fixed value, DPLL produces a *FailState*, returning the result that F is unsatisfiable [31].

Backjump

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

Chronological backtracking always goes back to the last decision literal l^d and changes 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 [31].

Learn

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

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 [31].

Forget

$$M \,\|\, F,\ C \quad\Longrightarrow\quad M \,\|\, F \quad \textbf{if} \big\{\ F \models C$$

Forget, on the contrary to **Learn**, removes lemmas with relevance or activity levels below a certain threshold. An activity could be, f.i., the number of times it becomes a unit or a conflicting clause. 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 [31].

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

Restart

$$M \,\|\, F \quad\Longrightarrow\quad \emptyset \,\|\, F$$

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 [31].

2.6.2 DPLL(T) paradigm

The DPLL(T) paradigm builds upon DPLL. However, before we can adapt our abstract DPLL model from above 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** this 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.

T-Learn

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

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** [31].

T-Forget

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

The only change in the **T-Forget** rule is that entailment between formulas becomes entailment in T [31].

T-Backjump

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

T-Backjump makes use of both, the propositional notion of entailment ($\models$) and the first-order notion of entailment modulo a theory ($\models_T$) [31].

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** will learn 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 [31]. Theoretically, that is how SMT solvers operate to solve SMT problems. However, most modern SMT solvers implement different methods to enhance performance. The most commonly used methods are described 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 [31].

On-line SAT solvers

After a T-inconsistency is detected and learned as a theory lemma, instead of beginning the search anew, the procedure will either apply **T-Backjump**, to go back to a point where the assignment was still T-consistent, or produce a *FailState* through the **Fail** rule [31].

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 [31]. **Theory propagation** is a kind of forward checking and plays an important role in DPLL(T). That is why, we add another rule to our abstract model from above.

TheoryPropagate

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

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 [31].

Chapter 3

SMT Solver Comparison

This chapter gives a brief overview of some state-of-the-art SMT solvers concerning their functionality. Only SMT solvers that can handle real numbers are considered, since they should be able to debug spreadsheets, and usually spreadsheets contain real numbers. Furthermore, the framework developed at the Graz University of Technology, which compares performance and execution time of different SMT- and constraint solvers when debugging spreadsheets, makes use of the MCSes-U algorithm that depends on the solvers' functionality to extract unsatisfiable cores. Therefore, it is important that the solvers support the extraction of unsatisfiable cores.

Table 3.1 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 3.2 and Table 3.3, whereas, a more detailed description follows below.

3.1 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. Further-

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 3.1: SMT solvers that support real numbers and participated in the SMT-COMP at least once [35], [3], [14]. Solvers shaded in gray are outdated. Either they are no longer in development or a newer version is available.

more, it is one of the few solvers able to handle every SMT-LIB logic [33]. 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 [14]. The number of benchmarks that could only be solved by Z3 increased to 21 benchmarks in 2013 [35]. 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 [13].

Technical characteristics

A more detailed description of Z3's characteristic features follows below [20], [19].

- **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

Name	Affiliation	Coding Language	License	API	Input Language	Models	Proofs	Unsat-Cores
CVC4 [26]	NYU, U. Iowa	C++	BSD	C++	SMT-LIB v1.0/v2.0, native language	yes	yes	no
MathSAT 5 [10]	U. Trento, FBK-irst	C++	Proprietary	C	SMT-LIB v1.2/v2.0, Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) format, native language	yes	yes	yes
SMTInterpol [9]	U. Freiburg	Java	LGPL v3	Java	SMT-LIB v1.2/v2.0, DIMACS format	yes	yes	yes
veriT [21]	U. Nancy, INRIA, UFRN	C	BSD	C	SMT-LIB v2.0, DIMACS format	yes	yes	no
Yices 2 [22]	SRI	C	Proprietary	C	SMT-LIB v1.2/v2.0, native language	yes	no	no
Z3 [20]	Microsoft Research	C++	MSR-LA	C, C++, .NET, Python, Java, OCaml	SMT-LIB v2.0, Simplify format, DIMACS format	yes	yes	yes

Table 3.2: 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 [28] 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 3.3: 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 gray denote the theories relevant for spreadsheet debugging.

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 [5].

- **Models and Proofs:** Microsoft's SMT solver is able to return models for satisfiable formulas. Furthermore, it can generate proofs for unsatisfiable formulas, and extract unsatisfiable cores. An unsatisfiable core is a subset of clauses of the input formula, which are T-unsatisfiable. Considering the SMT solvers supporting real numbers, Z3, SMTInterpol and MathSAT 5 currently are the only three solvers that support the functionality to extract unsatisfiable cores.

3.2 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 [26]. 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 [14]. In 2014's SMT-COMP CVC4 won in the divisions AUFLIA, AUFNIRA, LRA, QF_AUFBV, QF_LIA, QF_LRA, QF_UFNIA, UF and UFLIA [13].

Technical characteristics

CVC4 has four predecessors to learn from in terms of implementation architecture, and efficiency. Therefore, it offers many different features [26], [2].

- **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 [34]. 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.

3.3 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) [10]. 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 [3]. MathSAT 5, like SMTInterpol, was one of only 2 participants in the *unsat core track* of 2012's SMT-COMP [14]. 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 [13].

Technical characteristics

MathSAT 5 has been in constant development for many years to provide a vast array of functionality for its users [10].

- **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 Conflict-Driven Clause Learning (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 [11].

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: [11]

- 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.

3.4 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 [9]. 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 [14].

Technical characteristics

SMTInterpol is a very "young" SMT solver. However, it provides a wide range of features for its users [9].

- **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.

3.5 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 [6]. 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 [21]. 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 [6].

- **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 [21].

- **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.

3.6 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 [22]. 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 [3]. In 2014's SMT-COMP Yices 2 won the divisions QF_ALIA, QF_AUFLIA, QF_AX, QF_RDL, QF_UF and QF_UFBV [13].

Technical characteristics

To see what sets Yices 2 apart from the other solvers, its specifics are listed below [22].

- **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.

Chapter 4

Constraint Modeling Languages

The comparison of SMT solvers in Chapter 3 was conducted to find suitable candidates for an integration into the framework, to compare their performance and execution time when debugging spreadsheets. However, another important objective of the work is to find an efficient way to integrate the SMT solvers into the framework. That is why in this chapter we will present some constraint modeling languages and have a look at their applicability and range of application. Additionally, we will provide an example for each language to show the differences in their syntax.

4.1 MiniZinc

MiniZinc was created by the NICTA Optimisation Research Group with the goal to become a standard for the Constraint Programming (CP) community. Unlike above described SMT solvers, MiniZinc is a medium-level constraint modeling language, which is able to express most Constraint Satisfaction Problems(CSPs) easily. On the contrary to MiniZinc, FlatZinc is a low-level solver input language and the target language for MiniZinc. In other words this means that problems are formulated in MiniZinc and then converted into FlatZinc, before being passed to a solver capable of reading FlatZinc. The MiniZinc developers also offer an Integrated Development Environment (IDE), which supports users by writing MiniZinc models and allows them to run these models. Additionally, they hold a yearly competition called the MiniZinc Challenge, similar to the SMT-COMP. Until 2012 Gecode was the winner in all categories. However, in 2013, it only made third place in one category, where it was beaten by Opturion CPX and OR-Tools [25]. Unfortunately, MiniZinc and FlatZinc are barely supported by SMT solvers. However, there exists a compiler, called fzn2smt, which converts FlatZinc to the SMT-LIB language. Regrettably, the fzn2smt compiler only supports v1.2 of the SMT-LIB language and not the latest version. From the SMT solvers described in Chapter 3 only MathSAT 5, SMTInterpol and Yices 2 support the SMT-LIB v1.2 language. Therefore, FlatZinc, in combination with the fzn2smt compiler, could be used as an input language for these three solvers only.

4-Queens expressed in MiniZinc

To get a feeling of how MiniZinc looks like, let us have a look at an example. Example 4.1 shows the 4-Queens puzzle in the MiniZinc language. The first two lines define the number of queens to place on the board and the board itself. Line 3 sets the search strategy to search by selecting from the array *queens*, the variable with the currently smallest domain (*first_fail*), and try to set it to its median domain value (*indomain_median*), conducting a *complete* search. The constraints, stating that queens have to be placed in different rows, columns and diagonals, are defined in lines 4 to 6. The rest of the code is needed to generate the output.

4.2 SMT-LIB v2.0

SMT-LIB version 2.0 was specified by Cesare Tinelli, Clark Barett and Aaron Stump. On top of that specification Tinelli and Silvio Ranise created the SMT-LIB language. It was developed for the specific goal to create a common language across SMT solvers capable to express benchmark problems. Since Barrett and Stump together with Leonardo DeMoura also initiated the SMT-COMP, it was standing to reason to use the SMT-LIB language as the standard input language for the competition's benchmarks. This led to the result that many developers adapted this language into their SMT solver implementation, to be able to participate in the SMT-COMP [12]. Unlike MiniZinc, the SMT-LIB language itself is a low-level language. Meaning, it does not need to get translated, like MiniZinc gets

Example 4.1 4-Queens puzzle in MiniZinc [27]

```
int: n = 4;
array[1..n] of var 1..n: queens;          ▷ declaration of the chess board

solve :: int_search(                      ▷ sets search strategy
            queens,                       ▷ variables
            first_fail,                   ▷ chooses variable with smallest domain
            indomain_median,              ▷ assigns median domain values
            complete)                     ▷ conducts a complete search
      satisfy;                            ▷ indicates a satisfaction problem

constraint all_different(queens);         ▷ constraints
constraint all_different([queens[i]+i | i in 1..n]) :: domain;
constraint all_different([queens[i]-i | i in 1..n]) :: domain;

output [ show(queens) ++ "\n" ] ++        ▷ generate output
[
if j = 1 then
    "\n"
else
    ""
end if
++

if fix(queens[i]) = j then
    show_int(2,j)
else
    "___"
end if
| i in 1..n, j in 1..n
] ++ ["\n"];
```

converted into FlatZinc, since the solvers themselves support the language in its original form. All SMT solvers described in Chapter 3, as well as many other state-of-the-art SMT solvers, support the SMT-LIB v2.0 language.

4-Queens expressed in SMT-LIB v2.0

Again let us have a look at Example 4.2, which shows the 4-Queens puzzle in SMT-LIB v2.0, to get a feel for the syntax. In SMT-LIB v2.0 we first have to set a logic. This is done in the first line, where the logic is set to quantifier-free linear integer arithmetic. Since QF_LIA does not include the inequality operator $\neq$ it is defined in line 2. Lines 3 to 6 declare four different variables, each representing a queen of a specific row. For example, *q1* represents the position of the queen in the first row. *q2* the position of the queen in the second row, and so on. Lines 7 to 14 define the upper and lower bounds of the variable's value. The column constraints, stating that all queens have to be placed in different columns, are defined in lines 15 to 20. Lines 21 to 32 define the diagonal constraints. Finally, the **check-sat** function of line 33 tells the SMT solver to check for satisfiability. In case the formula is satisfiable, the **get-value** function of line 34 tells the solver to return valid values for the forwarded variables.

Example 4.2 4-Queens puzzle in SMT-LIB v2.0

```
(set-logic QF_LIA)                                   ▷ sets the SMT-LIB logic

(define-fun != ((x Int) (y Int)) Bool (not (= x y)))  ▷ definiton of the != operator

(declare-fun q1 () Int)              ▷ declaration of the 4 different variables
(declare-fun q2 () Int)
(declare-fun q3 () Int)
(declare-fun q4 () Int)

(assert (>= q1 1))                                        ▷ defines the bounds
(assert (<= q1 4))
(assert (>= q2 1))
(assert (<= q2 4))
(assert (>= q3 1))
(assert (<= q3 4))
(assert (>= q4 1))
(assert (<= q4 4))

(assert (!= q1 q2))                                    ▷ column constraints
(assert (!= q1 q3))
(assert (!= q1 q4))
(assert (!= q2 q3))
(assert (!= q2 q4))
(assert (!= q3 q4))

(assert (!= q1 (+ q2 1)))                     ▷ major diagonal constraints
(assert (!= q1 (+ q3 2)))
(assert (!= q1 (+ q4 3)))
(assert (!= q2 (+ q3 1)))
(assert (!= q2 (+ q4 2)))
(assert (!= q3 (+ q4 1)))

(assert (!= q1 (- q2 1)))                     ▷ minor diagonal constraints
(assert (!= q1 (- q3 2)))
(assert (!= q1 (- q4 3)))
(assert (!= q2 (- q3 1)))
(assert (!= q2 (- q4 2)))
(assert (!= q3 (- q4 1)))

(check-sat)                                      ▷ checks for satisfiability
(get-value (q1 q2 q3 q4))              ▷ returns valid values if satisfiable
```

Chapter 5

Conclusions and Future Work

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 paper we build upon the work conducted by a team from the Graz University of Technology where they introduced a framework to compare performance and execution time of SMT- and constraint solvers when debugging spreadsheets. However, within the scope of this work we solely focus on finding suitable candidates to expand their framework with modern SMT solvers. On these grounds we give a general overview of the topic SMT solvers, how they work and solve SMT problems. Furthermore, we conduct a comparison of 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 two, CVC4 and Z3, support 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 meets all our requirements. However, Z3 is already integrated in the framework. Therefore, CVC4, even though it is not able to extract unsatisfiable cores, is the most suitable candidate for an expansion of the framework. The integration of CVC4, however, can be conducted without great detriments, since next to the MCSes-U algorithm, the framework also contains an implementation of its predecessor, MCSes, which does not require unsatisfiable core extraction. Additionally, during our research, we found two constraint modeling languages for SMT solvers. However, we do not consider to use the MiniZinc modeling language in combination with the fzn2smt compiler, since fzn2smt supports only the SMT-LIB v1.2 language and the translation of the spreadsheet debugging problems to MiniZinc, FlatZinc and lastly to SMT-LIB v1.2 would result in a too complicated setup. Especially, when considering that all six described solvers of Chapter 3 are SMT-LIB v2.0 compliant, the SMT-LIB v2.0 modeling language seems to be the better choice. With an initial translation of the spreadsheet debugging problems into the SMT-LIB v2.0 modeling language, we will be able to easily expand the framework with the CVC4 SMT solver. Whether an initial translation, of the spreadsheet debugging problems into the SMT-LIB v2.0 language, results in a too great rise in execution time needs to be thoroughly tested. If so, the solver needs to be directly accessed via its API to get rid of this overhead. Finally, and most importantly, after the solver is integrated in the framework, a number of tests need to be conducted, to evaluate how it performs when debugging spreadsheets.

List of Figures

List of Tables

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.

DIMACS Center for Discrete Mathematics and Theoretical Computer Science.

DPLL Davis-Putnam-Logemann-Loveland.

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

FOL First-Order Logic.

GPL GNU General Public License.

IDE Integrated Development Environment.

LFSC Logical Framework with Side Conditions.

LGPL GNU Lesser General Public License.

MSR-LA Microsoft Research License Agreement.

NP	Nondeterministic Polynomial.
PL	Propositional Logic.
SAT	Boolean Satisfiability.
SMT	Satisfiability Modulo Theories.
SMT-COMP	Satisfiability Modulo Theories Competition.
SMT-LIB	Satisfiability Modulo Theories Library.

Bibliography

[1] S. Ausserlechner, S. Fruhmann, W. Wieser, B. Hofer, R. Spork, C. Muehlbacher, and F. Wotawa. The right choice matters! SMT solving substantially improves model-based debugging of spreadsheets. In Quality Software (QSIC), 2013 13th International Conference on Quality Software, pages 139–148, July 2013.

[2] Clark Barrett, Christopher L. 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.

[3] 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.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). Website, 2014. visited on July 11th 2014.

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

[6] Thomas Bouton, Diego Caminha B. 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.

[7] Roberto Bruttomesso. Satisfiability modulo theories: a pragmatic introduction. Lecture Notes, 2012.

[8] Roberto Bruttomesso, Alessandro Cimatti, Anders FranzÃ©n, 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.

[9] 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.

[10] 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.

[11] 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.

[12] David Cok. The SMT-LIBv2 language and tools: A tutorial. Tutorial, 2013.

[13] David Cok, David Deharbe, and Tjark Weber. SMT-COMP 2014. Website, 2014. visited on July 11th 2014.

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

[15] Jeremy Condit and Matthew Harren. Congruence closure. Lecture Notes, 2004.

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

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

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

[19] 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. CEUR-WS.org, 2008.

[20] 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.

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

[22] Bruno Dutertre. Yices 2 Manual. SRI International, Menlo Park, CA, March 2014.

[23] 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.

[24] Alessandro Farinelli. Propositional and first order logic. Lecture Notes, 2010.

[25] NICTA Optimisation Research Group. MiniZinc and FlatZinc. Website, 2014. visited on July 11th 2014.

[26] The ACSys Group. CVC4 User Manual. New York University and University of Iowa, 2014.

[27] Hakan Kjellerstrand. N-Queens problem in MiniZinc. Website - Example, 2014. visited on July 11th 2014.

[28] 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.

[29] 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.

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

[31] 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.

[32] Albert Oliveras and Enric Rodriguez-Carbonell. Combining decision procedures: The Nelson-Oppen approach. Lecture Notes, 2009.

[33] Microsoft Research. Z3 website. Website, 2014. visited on July 11th 2014.

[34] 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.

[35] Aaron Stump, Tjark Weber, and David Cok. Progress report on the 2013 SMT evaluation. Presentation Slides, 2013.

[36] Lintao Zhang. SAT-Solving: From Davis-Putnam to Zchaff and Beyond. Presentation Slides, 2009.