Master's Thesis

Suitability Analysis of CSP- and SMT-solvers for Test Case Generation

Hermann Felbinger, BSc

October 2013

supervised by:
Dr. Franz Wotawa. TU Graz, Austria
Dr. Ian Miguel. University of St. Andrews, United Kingdom
Dr. Christian Schwarzl. VIRTUAL VEHICLE Research Center, Austria

Graz University of Technology
Institute for Software Technology

Abstract

In Software testing it is demanded to uncover errors and corrupted behavior. This might be achieved by manually testing an implementation which requires a huge effort of man hours to perform the test cases. The discipline of model-based testing allows to generate test cases automatically by using paths through a model representing an implementation.
By using a Constraint Satisfaction Problem-solver or a Satisfiability Modulo Theory-solver in test case generation provides reasonable variable valuation but makes the time to generate the test cases highly dependable on the applied solver.
We picked a set of applicable solvers and generated a distinctive set of benchmarks to compare the solvers. For comparison we used the number of solved constraints and the time they consumed. The benchmarks represent a comparable input extracted from an Extended Symbolic Transition System. An Extended Symbolic Transition System represents a model for model based testing which was introduced as part of the STAte based system Test and simulatION tool suite at VIRTUAL VEHICLE Research and Test Center.
The results represent the differences of the applied solvers in their number of solved constraints and the time they consumed. These results and the reasons for the differences are discussed in this work.

Kurzfassung

Beim Testen von Software wird das Aufdecken von Fehlern und fehlerhaften Verhalten verlangt. Dies kann durch manuelles Testen einer Implementierung erreicht werden, was jedoch großen Aufwand an Mannstunden erfordert, das Testen durchzuführen. Die Disziplin des Modellbasierten Testens erlaubt das automatische Generieren von Testfällen mit Hilfe von Pfaden durch ein Modell, das die Implementierung abbildet.
Das Anwenden eines Constraint Satisfaction Problem-Solvers oder eines Satisfiability Modulo Theory-Solvers zur Testfallgenerierung bietet verwendbare Variablenwerte, ist jedoch bezüglich Laufzeit zur Generierung der Tests stark abhängig vom angewendeten Solver.
Wir wählten eine Zusammenstellung von anwendbaren Solvern und generierten eine ausgeprägte Menge an Benchmark-Tests um die Solver zu vergleichen. Für den Vergleich verwendeten wir die Anzahl der gelösten Constraints und die Laufzeit. Die Benchmark-Tests repräsentieren eine vergleichbare Eingabe, die aus einem Extended Symbolic Transition System extrahiert wurde. Ein Extended Symbolic Transition System repräsentiert ein Modell für Modellbasiertes Testen, das als Teil der STAte based system Test and simulatION Toolkette am VIRTUAL VEHICLE Research and Test Center eingeführt wurde.
Die Ergebnisse repräsentieren die Unterschiede der angewendeten Solver in Bezug auf die gelösten Constraints sowie auf die Laufzeit. Diese Ergebnisse und die Ursachen für deren Unterschiede werden in dieser Arbeit diskutiert.

Acknowledgement

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no 269335 (see Article II.9. of the JU Grant Agreement) and from the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economy, Family and Youth (BMWFJ) and the Austrian Research Promotion Agency (FFG).

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

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.

Graz, am ……………………………   ………………………………………………..
(Unterschrift)

Englische Fassung:

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.

……………………………   ………………………………………………..
date   (signature)

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contribution . . . 2
1.3 Outline . . . 2

2 Application **3**

3 Constraint Satisfaction Problem **5**
3.1 Basic concepts . . . 5
3.2 Unification . . . 6
3.3 Search Strategies . . . 6
3.3.1 Generate and Test . . . 6
3.3.2 Backtrack . . . 7
3.3.3 Backjumping . . . 7
3.3.4 Heuristics . . . 7
3.4 Constraint Propagation . . . 9
3.4.1 Node Consistency . . . 9
3.4.2 Arc Consistency . . . 10
3.4.3 Path Consistency . . . 10
3.4.4 Bounds Consistency . . . 11
3.4.5 k-Consistency . . . 12
3.5 Combinations of Search and Propagation . . . 12
3.5.1 Forward Checking . . . 12
3.5.2 Maintaining Arc Consistency . . . 12
3.6 Width of a CSP . . . 13

4 Satisfiability Modulo Theory **15**
4.1 Basic Concepts . . . 15
4.1.1 Conjunctive Normal Form . . . 16
4.1.2 DPLL(T) . . . 16
4.2 Satisfiability (SAT)-solver . . . 17
4.2.1 Stochastic Search . . . 17
4.2.2 DPLL-procedure . . . 18
4.3 Decision Procedures . . . 19
4.3.1 Propositional Logic . . . 20
4.3.2 Linear Integer Arithmetic . . . 20
4.3.3 Non-Linear Arithmetic . . . 21

4.4 Heuristics 21
4.4.1 Jeroslow-Wang 21
4.4.2 Dynamic Largest Individual Sum 22
4.4.3 Variable State Independent Decaying Sum 22
4.4.4 Berkmin 22
4.5 Theory Combination 23
4.5.1 Nelson-Oppen method 23
4.5.2 Model-based Theory Combination 23

5 Selected Tools 25
5.1 TreeSolver 25
5.2 Choco 26
5.3 Minion 26
5.3.1 Savile Row 26
5.4 Microsoft Z3 26
5.5 CVC4 27

6 Test Input 29
6.1 General Test Structure 29
6.2 Test Generation 30

7 Experimental Results 37
7.1 Environment 37
7.2 Setup 37
7.3 Results and Evaluation 39
7.3.1 Test results 39
7.3.2 Evaluation 53
7.4 Discussion 66

8 Related Work 69

9 Conclusion 71
9.1 Open Problems 71
9.2 Future Work 72

List of Figures

3.1 Width at an ordered node (41) 14
3.2 Width of a constraint graph (41) 14

4.1 Size of industrial Conjunctive Normal Form (CNF) formulas that are solved by SAT-solvers in a reasonable amount of time, according to the year (39). 17

7.1 Concept of the test environment 40
7.2 Results of running tests from the `boolean`-set 42
7.3 Results of running tests from the `+`-set 43
7.4 Results of running tests from the `random`-set 44
7.5 Barchart representing the number of solved tests for the '+'-set categorized in ranges of 400ms with formula length 1 47
7.6 Barchart representing the number of solved tests for the '+'-set categorized in ranges of 400ms with formula length 2 47
7.7 Barchart representing the number of solved tests for the '+'-set categorized in ranges of 400ms with formula length 3 48
7.8 Barchart representing the number of solved tests for the '+'-set categorized in ranges of 400ms with formula length 4 48
7.9 Barchart representing the number of solved tests for the '+'-set categorized in ranges of 400ms with formula length 5 49
7.10 Barchart representing the number of solved tests for the '*'-set categorized in ranges of 400ms with formula length 1 49
7.11 Barchart representing the number of solved tests for the '*'-set categorized in ranges of 400ms with formula length 2 50
7.12 Barchart representing the number of solved tests for the '*'-set categorized in ranges of 400ms with formula length 3 51
7.13 Barchart representing the number of solved tests for the '*'-set categorized in ranges of 400ms with formula length 4 51
7.14 Barchart representing the number of solved tests for the '*'-set categorized in ranges of 400ms with formual length 5 52
7.15 Barchart representing the number of solved tests for the '+'-set categorized by solver 52
7.16 Barchart representing the number of solved tests for the '-'-set categorized by solver 53
7.17 Barchart representing the number of solved tests for the '*'-set categorized by solver 54

7.18 Barchart representing the number of solved tests for the '/'-set categorized by solver . . . 54
7.19 Barchart representing the number of solved tests for the '%'-set categorized by solver . . . 55
7.20 Hypergraph representing CSP 1 . . . 58
7.21 Primalgraph representing CSP 1 . . . 59
7.22 Minimum width ordered constraint graph representing CSP 1 . . . 60
7.23 Hypergraph representing CSP 2 . . . 61
7.24 Primalgraph representing CSP 2 . . . 62
7.25 Minimum width ordered constraint graph representing CSP 2 . . . 63
7.26 Hypergraph representing CSP 3 . . . 64
7.27 Primalgraph representing CSP 3 . . . 64
7.28 Minimum width ordered constraint graph representing CSP 3 . . . 65

List of Tables

6.1 Logic operators 30
6.2 Arithmetic operators 30
6.3 Relational operators 31

7.1 Results for the test sets featuring time and number of passed tests . . . 45

1 Introduction

In the field of software testing we try to automate as much as possible. In all areas of software testing it is already usual to run tests automatically instead of hiring lots of software testers repeatingly executing the same tests. But not only the execution of tests should be automated, also the creation of the tests is desired to be done automatically.

Several different strategies for test case generation (model-based, mutation-based, specification-based, etc.) are currently investigated in parallel. These strategies rely on different approaches to generate test data. These approaches are in general either based on genetic algorithms, or use a Constraint Satisfaction Problem (CSP)-solver or a Satisfiability Modulo Theory (SMT)-solver which we investigate in this work.

1.1 Motivation

Test case generation using a CSP-solver or an SMT-solver stands and falls with the solver. The arithmetics contained in a model for which the test cases can be generated relies on the possibilities the solvers provide. The solvers have to able to handle the required inputs and reason about them as well as doing this in a reasonable amount of time. Considering this we restricted inputs for the solvers to First Order Logic (FOL)-formulas using arithmetic operations as functions, relational operators as predicates, and literals of types integer and boolean to create a reasonable input set to compare the different solvers.
For test case generation based on Extended Symbolic Transition System (ESTS)s a CSP- or SMT-solver which is able to treat FOL-formulas as CSPs and solves them in a reasonable time is required. These FOL-formulas in practice represent symbolic paths which describe a way through an ESTS. An executable test case requires feasible values. Therefore we need a valuation for the symbols in a symbolic path. The resulting valuation for a FOL-formula which is processed by a solver represents the test input for a System Under Test (SUT) which is represented by the ESTS.

1.2 Contribution

We compared the results of solvers for inputs of FOL structure. These results were either the time to solve if an input was solved correctly or an error message for the failed attempt. Using these results assisted us to find a solver which is best applicable for processing of input structured as a FOL-formula. Further we discussed the differences in the results and their root causes.

1.3 Outline

The reminder of this thesis is organized as follows: Chapter 2 provides a formal definition of ESTSs and its corresponding paths and traces. In this chapter also the application of the CSP- and SMT-solvers this work is targeting is introduced. Chapter 3 explains the preliminaries of constraint satisfaction and Chapter 4 explains the preliminaries of SMT which are used in this work.

Chapter 5 introduces the CSP-solvers and SMT-solvers which were applied in this work. The test inputs for these solvers to compare them and the procedure how these test inputs were generated is described in Chapter 6.

Chapter 7 shows the experimental results obtained by executing the solvers with the test inputs. The related work is discussed in Chapter 8. The thesis is concluded with a summary as well as with an outlook to future work in Chapter 9.

2 Application

We utilize a CSP- or SMT-solver during test case generation for integration testing. More precisely we use ESTSs to model a SUT and extract symbolic paths from these models which we use for test case generation. An ESTS is a behavioral model representing a component of the SUT. These ESTSs were introduced as part of the STAte based system Test and simulatION (STATION) tool suite at VIRTUAL VEHICLE Research and Test Center. As defined in (49) an ESTS is a tuple $<S, A, T, L, P, G, q_0>$ where S is a set of states, A is a set of attributes, T is a set of transitions, L is a set of labels, P is a set of signal parameters, G is a set of timing groups, and q_0 is the initial configuration. A symbolic path is a sequence of transitions which is created with respect to an ESTS, where a transition t $\in$ T is defined as (s, l, ϕ, ρ, p, d, s'). A transition t $\in$ T defines a state change where s represents the linked source state and s' the target state, l $\in$ L is a label, ϕ is a guard, ρ is an attribute update funcion, p is the priority of the transition, and d is its execution duration. The extraction of these symbolic paths from an ESTS is described in detail in (49). These extracted symbolic paths still contain the symbols of the attributes and signal parameters which are used in the guards and variable update functions. A guard represents a FOL formula which can either evaluate to true or false depending on the valuation of the variables. A guard which evaluates to true enables a state change. So a valuation which enables the state change is desired. The literals in these FOL-formulas are the signal parameters and the attributes which are both either of type integer or boolean. An attribute update function updates the attribute valuation after a state change within an ESTS. The concatenation of the guards and attribute update functions from the transitions in an execution path are called path constraint. A path constraint represents the input for the applied CSP- or SMT-solver which replies either a satisfiable valuation for the signal parameters or a message that the current execution path is not satisfiable and can not be applied as a test case for the SUT.

3 Constraint Satisfaction Problem

Constraints are present in our everyday life where some are natural e.g. the weather and some are man made e.g. the balance on everyones bank account. We have to deal with them permanently solving CSPs. This solving process is shown in how we try to live well within these constraints e.g. get warm clothes for frozen winter days with an affordable price that suits the balance on ones bank account.

3.1 Basic concepts

A combinatorial problem is modeled as a set of variables, representing the objects the problem deals with, and a set of constraints representing the relationships among the objects. Such a combinatorial problem is called a CSP (54).

More formal a CSP P, as defined by Eugene C. Freuder and Alan K. Mackworth in (26), is a triple $\mathtt{P} = \langle \mathtt{X}, \mathtt{D}, \mathtt{C} \rangle$ where $\mathtt{X}$ is an n-tuple of variables $\mathtt{X} = \langle \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \rangle$, $\mathtt{D}$ is a corresponding n-tuple of domains $\mathtt{D} = \langle \mathtt{D}_1, \mathtt{D}_2, \ldots, \mathtt{D}_n \rangle$ such that $\mathtt{x}_i \rightarrow \mathtt{D}_i$, $\mathtt{C}$ is a t-tuple of constraints $\mathtt{C} = \langle \mathtt{C}_1, \mathtt{C}_2, \ldots, \mathtt{C}_t \rangle$. A constraint $\mathtt{C}_j$ is a pair $\langle \mathtt{R}_{\mathtt{s}_j}, \mathtt{S}_j \rangle$ where $\mathtt{R}_{\mathtt{s}_j}$ is a relation between the variables in $\mathtt{S}_i = \mathtt{scope}(\mathtt{C}_i)$. The function $\mathtt{scope}(\mathtt{C}_i)$ returns the variables used in $\mathtt{C}_i$ what yields $\mathtt{R}_i$ as a subset of the Cartesian product of the domains of the variables in $\mathtt{S}_i$.

A solution of a CSP is an assignment of a value to each variable from its domain satisfying the constraints. In their paper Freuder and Mackworth (26) defined a solution to a CSP $\mathtt{P}$ as an n-tuple $\mathtt{A} = \langle \mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n \rangle$ where $\mathtt{a}_i \in \mathtt{D}_i$ and each constraint $\mathtt{C}_j$ is satisfied in that $\mathtt{R}_{\mathtt{s}_j}$ holds on the projection of $\mathtt{A}$ onto the scope $\mathtt{S}_j$.

Constraint Satisfaction is approached by the two basic algorithm groups constraint propagation (inferential consistency) and search. They can be applied separately but appear usually in integrated form to process a CSP. A third completely different ap-

proach is based on unification which origins in logic programming and is applied in one of the CSP-solvers in this work which is based on the PROLOG programming language.

3.2 Unification

In (51) the authors introduce unification as the basis of most work in automated deduction and of the use of logical inference in artificial intelligence. Unification uses a unifier of two terms which is a substitution that makes the terms identical. Two terms unify if they have a unifier.
An example for unification is an algorithm for unification of equations which takes as input two terms and replies either a unifier of the two terms or an error message if they do not unify. The used algorithms for the substitutions to solve constraints on boolean variables and linear arithmetic using finite domains are explained in detail in (32). A pioneering algorithm for satisfying constraints with unification is explained in (47). Example 1 shows a unification for a simple equation. If such a unification exists an equation is satisfiable otherwise not.

Example 1 (Unification of a simple equation)**.**
A unifier $\theta = \{\mathtt{x}_1/\mathtt{s}, \mathtt{x}_2/\mathtt{t}\}$ *subsitutes for an equation* $\mathtt{x}_1 = \mathtt{x}_2$ *variable* $\mathtt{x}_1$ *by* $\mathtt{s}$ *and* $\mathtt{x}_2$ *by* $\mathtt{t}$ *where* $\mathtt{s}$ *and* $\mathtt{t}$ *represent a valuation which satisfies the equation.* □

3.3 Search Strategies

The search for a solution to a CSP may be seen as exploring a tree where each node in the tree (except the root) corresponds to a unique variable assignment and each branch represents a partial assignment of the CSP. The common systematic search strategies are usually situated in a backtracking search. These search strategies emerged from several modified or substrategies of the Backtrack strategy.
In the worst case all search strategies to solve a CSP require exponential search time. The modified search strategies strongly reduce the probability that the worst case occurs.

3.3.1 Generate and Test

Generate and Test (GT) is an algorithm that generates randomly a valuation for each variable and consequently if this valuation satisfies all constraints in the CSP a solution

is found, otherwise another valuation is generated. It is a brute force method which checks constraints only after a complete assignment of all variables in the CSP. Its efficiency is poor because of the non-informed generator of the valuation and the late check for consistency, but it guarantees to find a solution if one exists. In GT in the worst case each possible complete assignment is generated to find a solution or to assess that the CSP is not satisfiable. This method is never used in practice but there exist some modified smarter generators using statistical approaches which improve the efficiency of GT as explained in (9). GT is usually not applied in practice.

3.3.2 Backtrack

Backtrack is a search method which guarantees to find a solution if a solution exists. It improves the GT method by incrementally extending partial solutions. This extending can be applied after checking if a constraint is violated after each assignment of a variable where all of the constraints are checked whether they are satisfied as soon as all of the variables they constrain are assigned. If a constraint is violated we have to backtrack. A very comprehensive introduction on backtrack and its efficiency is given in (36).

3.3.3 Backjumping

The backtrack search method retreats just one step backwards when encountering a dead-end. A dead-end occurs if there are no consistent values assignable to the current variable in the search tree relative to the current partial solution. This is improved by the backjumping method (46, 28) where the reasons for the dead-end are analyzed and irrelevant backtrack points can be avoided. This improvement leads to a jump back in the search tree directly to the root of the violating valuation. As backtrack tends to rediscover dead-ends repeatedly the backjumping method does not have this tendency.

3.3.4 Heuristics

A heuristic for solving a CSP includes a variable and/or a value selection procedure. The variable selection procedure should assign a value to that variable which maximally constrains the rest of the search space. The value selection procedure decides which value from the domain of a variable is most likely to lead to a satisfied constraint and should be assigned to the current variable. Current research also comprises constraint ordering heuristics as explained in (48) where the order in which constraints are solved

within a CSP may improve efficiency.

For the value selection procedure some common strategies are the min-value strategy which selects the minimum value, max-value which selects the maximum value, and mid-value which selects the median value in the remaining domain (1).

Usually value ordering strategies are not as important as variable ordering or variable selection strategies. Variable selection strategies are dynamic where the next variable which is valued is selected during runtime related to the chosen heuristic. A variable ordering is static and gives the variables in an order which defines the selection sequence as input to the CSP-solver. Variable selection strategies are for example the *wdeg* and *dom over wdeg* (11). The *wdeg* strategy selects the variable which is mostly involved in violated constraints which are weighted. The constraints are weighted depending how often they were violated in the so far arranged search. So applying the *wdeg* strategy selects the variables depending on the maximal weight of the constraints where they occur. The *dom over wdeg* heuristic involves the domain of the variables in the selection procedure. In this heuristic the size of the domain (*dom*) of each variable and the value of the *wdeg* are used to calculate a ratio $\frac{dom}{wdeg}$. The next selected variable in this strategy is the variable with the minimum ratio.
Further dynamic variable selection heuristics are *dom-deg*, *min-dom*, and *max-dom*. The *dom-deg* heuristic is similar to *dom over wdeg* strategy but uses only a counter of variable occurrences (*deg*) in constraints without weighting them. The next selected variable in the *dom-deg* heuristic is the variable with the minimum ratio of $\frac{dom}{deg}$. The *min-dom* heuristic chooses the variable with the smallest domain and *max-dom* chooses the variable with the largest domain next.

In (22) Rina Dechter shows a classification of the heuristics which are used to improve backtrack during search. She introduces lookahead schemes and lookback schemes where variable selection and value ordering appertain to lookahead schemes. Lookback schemes are e.g. go back to source of failure and constraint recording. Constraint recording can be applied if a backtracking algorithm encounters a dead-end. The dead-end rises if the current partial valuation of the variables is consistent but there is no consistent value in the domain of the next variable. This partial valuation is recorded as an additional constraint to make sure that this valuation is not revisited during search. A constraint ordering heuristic applicable on constraint recording is explained in (48).

3.4 Constraint Propagation

A comprehensive overview of the roots of constraint propagation and the different algorithms to implement constraint propagation can be found in (40) (This paper of Alan K. Mackworth was honored in Artificial Intelligence 59, 1-2, 1993 as one of the fifty most cited papers in the history of Artificial Intelligence). Constraint propagation prunes the search tree by removing inconsistent valuations from the domains of the variables by deduced information which is recorded as change to the problem. A constraint is consistent if it reaches a point where nothing new can be deduced. Inconsistent valuations are detected by running different local and global consistency checks. The consistency checks which are used in this work are Node Consistency, Arc Consistency, Path Consistency, Bounds Consistency, and k-Consistency. The CSP-solvers applied in this work use Node Consistency and Arc Consistency. Local consistency checks treat constraint by constraint within a CSP to achieve local consistency whereas to achieve global consistency any consistent instantiation of a subset of the variables can be extended to a consistent instantiation of all the variables without encountering any dead-ends.

3.4.1 Node Consistency

Node consistency checks are applied on unary constraints. Local node consistency holds if for every value in the domain D_i for variable x_i the unary constraint $c(x_i)$ is satisfied. If domain D_i contains values which do not satisfy the constraint $c(x_i)$ these values are removed to gain node consistency. An unary constraint $c(x_i)$ is not satisfiable if the domain D_i is empty after applying a node consistency check. Global node consistency can be achieved if local node consistency holds for all unary constraints. Example 2 shows the application of a node consistency check.

Example 2 (An unary constraint showing the application of the node consistency check).
variables $X = \{x_1\}$
domain $D_1 = \{3, 4, 5, 6\}$
constraint $c(x_1)$ *represents* $x_1 < 5$

The application of node consistency on c *removes the values* $5, 6$ *from domain* D_1 *where the remaining values in* $D_1 = \{3, 4\}$ *are node consistent.*

□

3.4.2 Arc Consistency

Arc consistency checks are applied on binary constraints. Local arc consistency holds if the two variables x_i and x_j which are constrained by the binary constraint $c(x_i, x_j)$ are both node consistent and there is at least one value a_j in the domain D_j for the variable x_j such that the constraint $c(x_i, x_j)$ is satisfied if a value $a_i \in D_i$ for variable x_i is assigned to x_i. This consistency check is applied in both directions. As explained there is at least one value for x_j for every value in the domain of x_i and there is at least one value for x_i for every value in the domain of x_j. If there is no value $a_j \in D_j$ of variable x_j then the currently assigned value to x_i has to be pruned. Global arc consistency holds if local arc consistency holds for all binary constraints in a CSP. Simply enforcing local consistency for all constraints to achieve global arc consistency is not very efficient because for every change in the domain of a variable all binary constraints have to be rechecked to be arc consistent. To achieve global arc consistency huge efforts were made in developing algorithms AC1, AC2, AC3, AC4, AC6, and AC3.1 which are explained in detail in (21) their differences are in worst-case time complexity and worst-case space complexity. Example 3 shows the application of an arc consistency check.

Example 3 (A binary constraint showing the application of the arc consistency check)**.**
variables $X = \{x_1, x_2\}$
domains $D_1 = D_2 = \{3, 4, 5, 6\}$
constraint $c(x_1, x_2)$ *represents* $x_1 < x_2$

The application of arc consistency on c *removes value* 6 *from domain* D_1 *where the remaining values in* $D_1 = \{3, 4, 5\}$ *are arc consistent and removes value* 3 *from domain* D_2 *where the remaining values in* $D_2 = \{4, 5, 6\}$ *are arc consistent.* □

3.4.3 Path Consistency

Local path consistency can extend an arc consistent assignment to any two variables to a three variables consistency. Thus path consistency checks are applied on binary constraints which have a relation to a third variable. A constraint $c(x_i, x_j)$ is path consistent relative to a third variable x_k if every valuation a_i of x_i and a_j of x_j from their domains D_i and D_j satisfy the constraint $c(x_i, x_j)$ and there also exists a value a_k from domain D_k for variable x_k such that the values in D_i and D_k satisfy a constraint $c(x_i, x_k)$ and the values $a_j \in D_j$ and $a_k \in D_k$ satisfy a constraint $c(x_j, x_k)$. The assignments are always expressed in tuples e.g. $\langle a_i, a_j \rangle$. So if no value $a_k \in D_k$ for x_k exists such that $\langle a_i, a_k \rangle$ satisfies $c(x_i, x_k)$ and $\langle a_j, a_k \rangle$ satisfies $c(x_j, x_k)$ then we can remove the values

$\langle a_i, a_j \rangle$ which satisfy $c(x_i, x_j)$ from the domain of x_i and x_j to achieve path consistency. If a domain becomes empty the CSP is not satisfiable. Another approach to enforce path consistency is to add constraints during runtime to the CSP instead of removing values from the domains. In Example 4 instead of removing value 6 from domain D of variable x we just can add a constraint $c(x)$ where $c(x)$ represents $x \neq 6$.
Global path consistency holds if for every three distinct variables x_i, x_j, x_k within a CSP a constraint $c(x_i, x_j)$ is path consistent relative to a variable x_k. Algorithms for enforcing path consistency like PC1 and PC2 are similar to those enforcing arc consistency. Path consistency does not basically imply arc consistency. Example 4 shows the application of a path consistency check.

Example 4 (A binary constraint showing the application of the path consistency check).
variables $X = \{x_1, x_2, x_3\}$
domains $D_1 = [0..4]$, $D_2 = [1..5]$, $D_3 = [5..10]$
constraint $c_1(x_1, x_2)$ *represents* $x_1 < x_2$
constraint $c_2(x_2, x_3)$ *represents* $x_2 < x_3$
constraint $c_3(x_1, x_3)$ *represents* $x_1 < x_3$

The given CSP is not path consistent because there is no value $a_2 \in D_2$ *for variable* x_2 *where* $4 < x_2 < 5$ *if* $x_1 = 4$ *and* $x_3 = 5$. *The application of path consistency on* c_1, c_2, c_3 *removes value* 5 *from domain* D_3 *where the remaining values in* $D_3 = [6..10]$ *are path consistent. Another opportunity is to add a further constraint* $c_4(x_3)$ *which constrains* x_3 *to* $x_3 \neq 5$. □

3.4.4 Bounds Consistency

The bounds of a variable are the minimum value and the maximum value of its related domain. A constraint is bound consistent if for any variable in the constraint each of its bounds can be extended to a tuple satisfying the constraint. In other words the check for bounds consistency requires that the minimum value and the maximum value of the domain of a variable can be used to satisfy the constraint where the other variables are valued with any value within their domain. Bounds consistency is used when the domain of a variable is large and represents an interval and is achieved by tightening the bounds of the domain by removing unsupported minimum and maximum values. Bounds consistency is rather effective in terms of operations to achieve consistency than applying global arc consistency (21).

3.4.5 k-Consistency

k-consistency (also called i-consistency) is given if an assignment to k-1 variables satisfies all constraints using them and choosing an assignment to any k-th variable satisfies all constraints applying these k variables. k-consistency does not require binary CSPs which only contain constraints of arity 2. Enforcing k-consistency does not remove values from a domain of a variable like enforcing arc consistency but adds constraints to the CSP which rule out values from a domain of a variable. These additional constraints are called *no-goods*.
Strong k-consistency is given if we have j-consistency for all $1 \leq j \leq k$. In other words if we have a consistent assignment for j-1 variables and this assignment can be extended to a consistent assignment for j variables for all $1 \leq j \leq k$. Strong k-consistency has the characteristic that you never have to backtrack while searching for a CSP solution and is therefore backtrack-free. On the other hand to enforce strong k-consistency becomes very quickly very expensive thus k-consistency is very rarely implemented in the popular CSP-solvers.

3.5 Combinations of Search and Propagation

Typically CSP-solvers use a combination of a search method and constraint propagation. There are lots of possibilities to combine these methods. Two of the most common combinations are forward checking and maintaining arc consistency.

3.5.1 Forward Checking

A combination of search and constraint propagation is forward checking. It represents the fundamental way how systematic constraint solvers work. Forward checking first guesses an assignment for a variable, then propagates all consequences arisen by this assignment and continues again with a guessing if no constraint is violated and the CSP is not satisfied yet. Forward checking guarantees to explore a smaller search tree than backtracking.

3.5.2 Maintaining Arc Consistency

A combination of constraint propagation and search using global arc consistency is called Maintaining Arc Consistency (MAC) where arc consistency is embedded in a

search algorithm. This combination spots dead-ends in a search tree earlier than forward checking reducing the search effort (10). MAC does not need to check backwards as well. The difference to forward checking is that forward checking checks each unassigned variable separately whereas MAC applies global arc consistency.

3.6 Width of a CSP

As Eugene C. Freuder clarified in (27) a valuation for a variable order is backtrack-free if the level of strong k-consistency exceeds the width of the correspondingly ordered constraint graph. Arc consistency is defined as strong k-consistency for k = 2 and strong path consistency is strong k-consistency for k = 3.

The width of a variable order is the maximum width of a node in an ordered constraint graph where an ordered constraint graph is an arrangement of nodes into a fixed linear order. The minimum width of a constraint graph is the minimum width of all of its orderings. Each node in a constraint graph represents a variable in the CSP. The width of a node in a constraint graph is the number of connections to past resp. overlying nodes in the constraint graph. Figure 3.1 shows the width of a node $\{x_1 = 0, x_2 = 3, x_3 = 1, x_4 = 2, x_5 = 1\}$ and the width of the constraint graph 3 which is the maximum width of a node within the graph. An example to show the different possible constraint graphs to get the minimum width of the constraint graph out is shown in Figure 3.2. The minimum width of the CSP in Figure 3.2 is 1 because it is the minimum width of all possible orderings. The width of the constraint graphs is denoted below each constraint graph in Figure 3.2 and the width of the nodes is given right to the nodes.

Algorithm 1 shows the greedy Min-Width algorithm to sort an ordered constraint graph to a minimum width ordered constraint graph.

Algorithm 1 Min-Width (21)

Input: A graph $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$.
Output: A min-width ordering of the nodes $d = (v_1, \ldots, v_n)$.

1: **for** $j = n$ to 1 by -1 **do**
2: $r \leftarrow$ a node in G with smallest degree.
3: Put r in position j and $G \leftarrow G - r$.
(Delete from V node r and from E all its adjacent edges)
4: **end for**

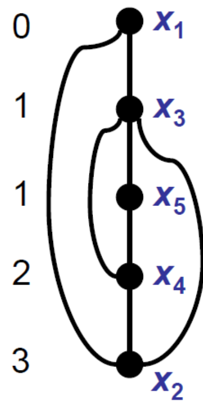

Figure 3.1: Width at an ordered node (41)

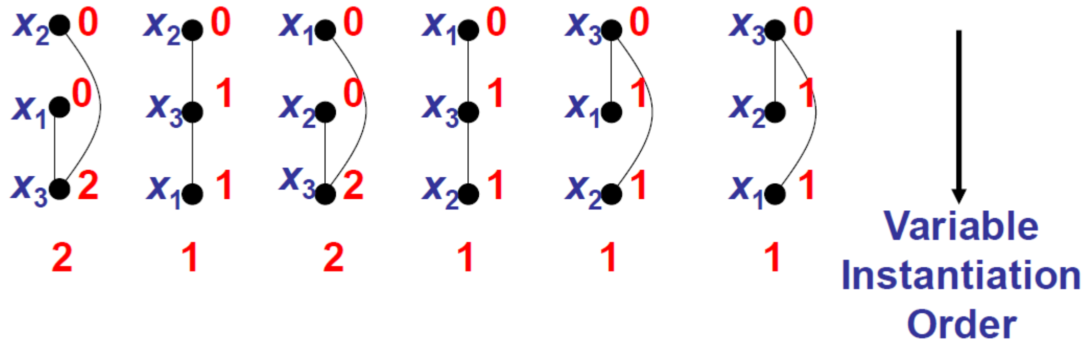

Figure 3.2: Width of a constraint graph (41)

4 Satisfiability Modulo Theory

Despite propositional logic applications, which are processed by SAT solvers more expressive logics like FOL are investigated in computer science and mathematics. Many applications to investigate satisfiability require not just general FOL satisfiability, but rather satisfiability with respect to some background theory which interprets certain predicate and function symbols. The research field concerned with the satisfiability of formulas with respect to some background theory is called Satisfiability Modulo Theory.

4.1 Basic Concepts

SMT-solvers combine the problem of boolean satisfiability with domains (such as those studied in convex optimization (12)). They involve investigations in the decision problem, completeness and incompleteness of logical theories, and the complexity theory (19). SMT relies on existing decision procedures that know how to reason about a particular theory, what means that SMT is less generalized than convex optimization e.g. that the "+"-operator behaves as expected as shown in Example 5.

Example 5 ("+"-operator).
the "+"-operator in linear arithmetic is assumed to behave the way we learn it at elementary school: we don't have to specify that $\forall \mathtt{x}, \mathtt{y}.\ \mathtt{x} + \mathtt{y} = \mathtt{y} + \mathtt{x}$ □

The major approaches so far for implementing SMT-solvers are usually referred to as eager, lazy, and DPLL(T) approach (8). The eager approach converts the input formula in an equisatisfiable propositional formula before checking the satisfiability. This approach is extensively explained in the papers (45, 52, 50, 14, 15) which were published in the years 1999-2003. The lazy approach consists in building ad-hoc procedures implementing an inference system specialized on a background theory. This approach is extensively explained in the papers (2, 20, 5, 3) which were published in the years 2000-2002.

4.1.1 Conjunctive Normal Form

The input of an SMT-solver is usually preprocessed and transfered in a boolean structure for which the internal SAT-solver is designed to work with. The most common boolean structure is the CNF. A formula in CNF is a conjunction of clauses where each clause is a disjunction of literals. A literal represents an atom in a propositional formula or the negation of an atom. In the remainder of this work we name literals in propositional logic formulas also boolean variables. To gain a propositional formula from a FOL formula which is the input of an SMT-solver an abstraction step is required. This step abstracts away the functions and predicates of an FOL formula and translates them into boolean variables. This new formula can be transfered in CNF and processed by a regular SAT-solver (19).

4.1.2 DPLL(T)

DPLL(T) is the latest approach for implementing SMT-solvers. It is an efficient and modular approach based on a general DPLL(X) engine where X can be instantiated with a specialized theory solver T to produce a DPLL(T) system. In other words the combination of a Davis-Putnam-Logemann-Loveland (DPLL) based SAT-solver and a theory solver results in a DPLL(T) based SMT-solver. In this approach the SAT-solver chooses literals from an input which is usually transferred into CNF and assigns a valuation. The valuation of these literals is crucial for the satisfiability of the boolean structure. The theory solvers check whether the chosen variables and their valuations are consistent in the applied theory. The valuation is consistent if the function which is abstracted by the literal evaluates to the assigned valuation of the literal. Improved versions of the DPLL(T) invoke the theory solver after a partial assignment of the SAT-solver instead of waiting for a full assignment. A partial assignment gives the possibility to derive implications for the SAT-solver after the theory solver finishes. An Example for such an implication is shown in 6. This implications are also called theory propagation. For a comprehensive introduction of the DPLL(T) approach we refer the reader to (44).

Example 6 (Implication of partial assignment).
The abstraction of an input $\texttt{x} = \texttt{z}$ *is a boolean variable* `p`. *A SAT-solver assigns the value* `true` *to* `p` *and invokes the theory solver which in this case is a theory solver for equality logic with uninterpreted functions.*
The theory solver assigns a valuation to $\texttt{x} = \texttt{z}$ *which makes* `p` `true`. *A derived implication for the SAT-solver of this assignment is that another boolean variable* `q` *which is an abstraction of* $\texttt{x} \neq \texttt{z}$ *must be* `false`. □

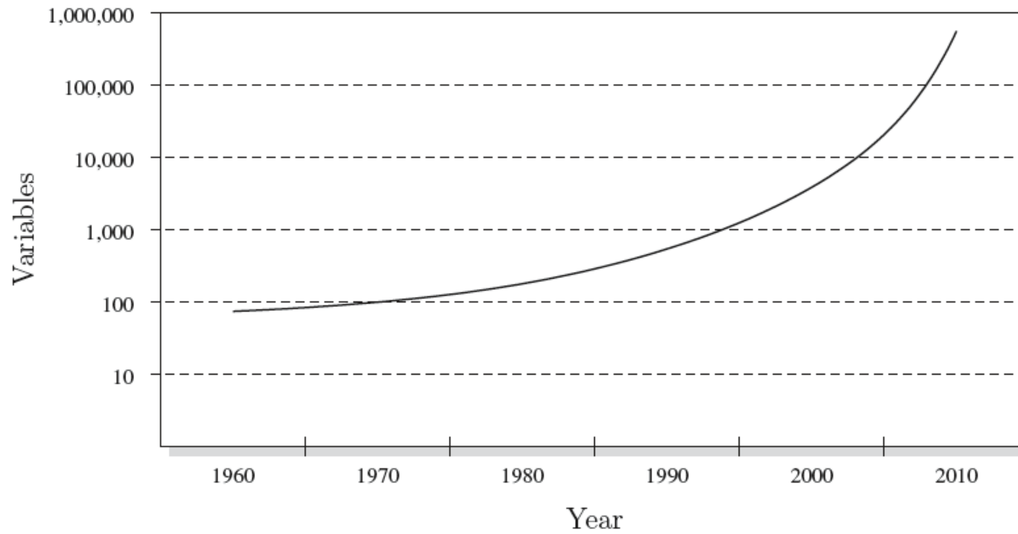

Figure 4.1: Size of industrial CNF formulas that are solved by SAT-solvers in a reasonable amount of time, according to the year (39).

4.2 SAT-solver

For SMT-solvers a SAT-solver represents the core which handles propositional formulas. SAT-solvers aim to decide whether a propositional logic formula which uses only boolean variables can be made true by assigning a valuation. This valuation is either true or false to each of the formulas variables. Modern SMT-solvers use a SAT-solver which solves formulas in CNF. These modern SAT-solvers can solve formulas in CNF with hundreds of thousands of variables in a reasonable amount of time of a few hours (39). Figure 4.1 shows the rise of numbers of variables in a propositional logic formula in CNF that are regularly solved by SAT-solvers in a reasonable amount of time.

These SAT-solvers can be grouped in two main categories. The first are based on stochastic search and the second category of SAT-solvers are based on the DPLL procedure (16).

4.2.1 Stochastic Search

SAT-solvers based on stochastic search are guessing an initial assignment and check this assignment whether it satisfies the input formula. If this check evaluates to false the SAT-solver starts to change the valuations of variables according to an internal heuristic. After each change a check whether the input formula is satisfied follows. The

heuristic is typically based on counting the number of unsatisfied clauses and chooses the variable to change which minimizes this number.

4.2.2 DPLL-procedure

DPLL (16) is a procedure which makes decisions about the valuation of a variable, propagates implications after a decision, and backtracks if a conflict occurs. This procedure can be applied on boolean variables and therefore seen as a procedure acting on a binary tree. Each level in the binary tree represents a decision level. The root node in the binary tree has decision level 1 because the first decision is made there.

Algorithm 2 shows the DPLL-procedure. The main components in this algorithm are the functions: DECIDE(), BCP(), ANALYZE-CONFLICT(), BACKTRACK(dl).

Algorithm 2 DPLL-SAT (39)

Input: A propositional CNF formula $\mathfrak{B}$
Output: "Satisfiable" if the formula is satisfiable and "Unsatisfiable" otherwise

```
 1: function DPLL
 2:     if BCP() = "conflict" then return "Unsatisfiable";
 3:         while TRUE do
 4:             if ¬DECIDE() then return "Satisfiable";
 5:             else
 6:                 while BCP() = "conflict do
 7:                     backtrack-level := ANALYZE-CONFLICT();
 8:                     if backtrack-level < 0 then return "Unsatisfiable";
 9:                     else
10:                         BackTrack(backtrack-level);
11:                     end if
12:                 end while
13:             end if
14:         end while
15:     end if
16: end function
```

DECIDE() chooses a so far unassigned variable and a valuation for it. BCP() applies boolean constraint propagation (unit propagation) until either a conflict is encountered or no more implications are possible. ANALYZE-CONFLICT() detects unsatisfiability of the input formula with the current instantiation of the variables and computes the backtrack level. The backtrack level is the level in the binary tree where to backtrack to in case of a conflict. Further the ANALYZE-CONFLICT() function adds new constraints to the input formula during runtime which constrain the search tree

after a conflicting assignment. BACKTRACK($backtrack - level$) jumps to the level $backtrack - level$ in the binary tree and erases assignments at all higher levels.

SMT-solvers got more and more attention because of the improvements in the performance of SAT-solvers based on the DPLL procedure in the last years. This improvements were made by better implementation techniques like the two-watched literal approach for unit propagation. Further they were made by enhancements on the reduction of the search space like backjumping, conflict-driven lemma learning, and restarts as explained in (44).

SAT-solvers based on DPLL are considered to be better in terms of the number of formulas which can be solved in a reasonable amount of time. The SMT-solvers applied in this work use SAT-solvers which are based on the DPLL-procedure.

4.3 Decision Procedures

A decision procedure is an approach to decide whether an input formula is satisfiable. A satisfiable formula requires at least one assignment to the used variables which let the formula evaluate to true. An assignment of a variable is a mapping of the variable to a value in its domain where the domain represents all values which can be assigned to this variable. Every variable has a related domain. If a formula is satisfied under each possible assignment the formula is called valid. A formula which is not satisfiable is named as contradiction. Decision procedures are based on deduction and enumeration but not exclusivly on either deduction or enumeration. An example for a decision procedure based on enumeration is a truth table for a propositional logic formula. An example for a deduction based decision procedure is the application of the contradiction rule. The contradiction rule derives a conclusion by an inference which says that the propositional logic formula $(\mathtt{x} \wedge \neg\mathtt{x})$ always evaluates to false. This evaluation result can be deduced and no more processing is required.

Decision procedures are different for different theories. The different theories which were applied in this work are the theories for propositional logic, linear integer arithmetic, and nonlinear integer arithmetic. The decision procedures for linear and nonlinear integer arithmetic comprise of the decision procedures for equality and difference arithmetic as well. The decision procedure which is applied by the SMT-solver depends on the structure of the input formula. Depending on the SMT-solver these decision procedures are either selected automatically or manually by the user.

4.3.1 Propositional Logic

A decision procedure for propositional logic is applied by a SAT-solver which is explained in Section 4.2. This decision procedure decides whether a formula in propositional logic is satisfiable.

4.3.2 Linear Integer Arithmetic

A linear integer arithmetic constraint is satisfied if the valuation of each variable in the constraint is a value from the set of integers and evaluating the constraint to true. A famous decision procedure for solving linear integer arithmetic constraints is the Branch and Bound algorithm which was initially used for numerical optimization. An slightly modification of Branch and Bound can be used as a decision procedure for linear integer arithmetic constraints. This Branch and Bound algorithm applies a decision procedure which can handle a relaxed version `relaxed(c)` of the input CSP `c`. The relaxed version omits the requirement that the valuation for the variables must be a value from the set of integers. An example for such a decision procedure is the Simplex algorithm which decides if the input constraint is either satisfiable or not. If the valuation for the satisfied `relaxed(c)` after applying the Simplex algorithm contains non integer values the problem is split in two subproblems. This splitting derives from the Branch and Bound approach. Otherwise the decision procedure returns that the input CSP is satisfied. The splitting adds a constraint for a non integer valuation of a variable $\mathtt{x_i}$ such that $\mathtt{x_i} \leq \lfloor \mathtt{current\ valuation} \rfloor$ to the CSP `c`. Then a recursive call of the Simplex algorithm follows with input `c`. This splitting and the recursive call is repeated until either a valuation consisting of values from the set of integers is found or the CSP is not satisfiable. If the CSP is not satisfiable with the additional constraint then the CSP is augmented with the constraint $\mathtt{x_i} \geq \lceil \mathtt{current\ valuation} \rceil$. Again this splitting and recursive call is applied until either a satisfying valuation consisting of values from the set of integers is found or the CSP is not satisfiable. If no valuation consisting of values from the set of integers can be found the CSP is not satisfiable for linear integer arithmetic. A full overview is of the Branch and Bound approach is given in (39).
The Simplex algorithm is an algorithm for numerical optimization which can be applied to solve linear constraints over variables of type real in a modified version. The Simplex algorithm is based on geometrical research. In geometrical terms each variable of a CSP corresponds to a dimension and each constraint defines a subspace. A satisfying valuation can be found in the intersection of these subspaces. An application of the Simplex algorithm within an SMT-solver is explained in detail in (18). Another newer approach to solve linear integer arithmetic constraints using the Simplex algorithm is introduced in (34). As mentioned the Simplex algorithm adds a dimension for each variable in the CSP. In (34) they introduce an approach where they eliminate a part of

the search tree if a non integer valuation for a CSP is found by deriving planes within this geometrical space constructed by the variables which represent the non integer valuation. Due to the fact that a solution for the Simplex algorithm is an intersection of subspaces the deriving of planes which represent valuations which do not lead to a desired solution prune the search space substantial. The approach of adding planes is called the Cutting Planes approach.

4.3.3 Non-Linear Arithmetic

A non linear arithmetic constraint for integer valuations is not decidable. Therefore a decision procedure for real valuation can be applied to find a valuation which can facilitate the search for an integer valuation. A famous approach is the Cylindrical Algebraic Decomposition (CAD). This approach works by decomposing $\mathbb{R}^k$ into connected components such that all of the polynomials from the CSP are invariant regarding their sign. This is possible because CAD first performs a projection of the polynomials from the initial problem. Such a projection contains many new polynomials which are derived from the initial polynomials. These polynomials contain enough information to ensure that the decomposition is possible. The size of these projection sets depends on the number of variables. In (35) an improved approach applying CAD is introduced.

4.4 Heuristics

There is a vast number of heuristics known which define the selection order of variables and values which are assigned next. In SMT-solvers this heuristics are the same as for SAT-solvers because they consider the boolean abstraction of the input formula as explained in Section 4.1.1. Some common heuristics are Jeroslow-Wang, Dynamic Largest Individual Sum (DLIS), Variable State Independent Decaying Sum (VSIDS), and Berkmin.

4.4.1 Jeroslow-Wang

The Jeroslow-Wang heuristic requires as input a propositional formula in CNF to calculate a value $\mathtt{J(l)}$ for every variable $\mathtt{l}$. In (39) the formula to calculate $\mathtt{J(l)}$ for each variable $\mathtt{l}$ in a propositional formula $\mathfrak{B}$ is given as

$$\mathtt{J(l)} = \sum_{\omega \in \mathfrak{B}, \mathtt{l} \in \omega} 2^{-|\omega|}$$

where ω represents a clause and $|\omega|$ the length of a clause. In this heuristic the variable with the maximum value for `J(l)` is chosen. This heuristic gives higher priority to variables that appear frequently in short clauses.

4.4.2 Dynamic Largest Individual Sum

In the DLIS heuristic an unassigned variable that satisfies the largest number of currently unsatisfied clauses is chosen next. To get this number for each variable a list of references to clauses in which a variable appears has to be kept. The number of unsatisfied clauses can be extracted by counting the references of each variable to unsatisfied clauses. This imposes a large overhead to apply his heuristic.

4.4.3 Variable State Independent Decaying Sum

The VSIDS heuristic is very similar to DLIS but does not regard the question if a clause is already satisfied for the number of clauses in which a variable appears. The number of clauses in which a variable appears is kept for each variable and labeled as score. This score is initially the number of clauses in which a variable appears but is divided by a constant (e.g. 2) after a variable was chosen. If a conflict occurs and a clause cannot be satisfied with the current valuation this conflict is added as a new constraint. This new constraint adds the value 1 to the score of each variable which appears in the new constraint. Thus variables in these new constraints become more influential than others.

4.4.4 Berkmin

The Berkmin heuristic is very similar to VSIDS but does not divide the score for each variable after a variable was chosen. In comparison to VSIDS it only considers unresolved conflicts which causes new constraints. In this heuristic a new constraint added due to a conflict is pushed on a stack. If a variable has to be chosen the first constraint on this stack which was added because of a still unresolved conflict is chosen and the variable within this constraint with the highest score is selected. If the stack is empty the variable with the overall highest score is selected. In this heuristic variables which appear in recent conflicts have absolute priority.

4.5 Theory Combination

Decision procedures are limited on specific functions. In practice different theories can be applied in combination in a single CSP. Therefore also for an SMT-solver combinations of the decision procedures are required to handle these CSPs. The two most famous theory combination methods are the Nelson-Oppen method and a model-based theory combination.

4.5.1 Nelson-Oppen method

The Nelson-Oppen method is explained in (42). It provides a method to combine theories but with a few restrictions which are:

1. $T_1, \ldots, T_n$ are quantifier free FOL theories with equality.

2. There is a decision procedure for each of the theories $T_1, \ldots, T_n$.

3. The signatures are disjoint, i.e., for all $1 \leq i < j \leq n$, $\sum_i \cap \sum_j = \emptyset$.

4. $T_1, \ldots, T_n$ are theories that are interpreted over an infinite domain (e.g. linear arithmetic over $\mathbb{R}$, but not the theory of finite-width bit vectors).

A signature of a theory is a set of function of predicate symbols over which a theory is defined. A convex theory has for every satisfiable set of literals a model where variables not implied to be equal have a distinct interpretation.
These restrictions and the algorithm shown in Listing 3 are introduced in (39). The algorithm is a procedure for combinations of convex theories. The input formula must be a conjunction of literals. The first step in 3 purifies each literal so that it belongs to a single theory by replacing the literals with auxiliary variables adding a constraint which assures that the literal equals the auxiliary variable. This results in a set of pure expressions $F_1, \ldots, F_n$ which represent these equality constraints. Only disjunctions of these pure equality expressions are communicated by the theory solvers.

4.5.2 Model-based Theory Combination

Model-based theory combination is introduced in (17). Their approach minimizes the number of shared equalities which are explained in Section 4.5.1. This benefits from the fact that in practice the number of local inconsistencies is much bigger than the number of inconsistencies across theories. Model-based theory combination does not

Algorithm 3 NELSON-OPPEN-FOR-CONVEX-THEORIES (39)

Input: A convex formula ϕ that mixes convex theories

Output: "Satisfiable" if ϕ is satisfiable, and "Unsatisfiable" otherwise

1: Purification: Purify ϕ into $F_1, \ldots, F_n$.

2: Apply the decision procedure for T_i to F_i. If there exists i such that F_i is unsatisfiable in T_i, return "Unsatisfiable".

3: Equality propagation: If there exist i, j such that F_i T_i-implies an equality between variables of ϕ that is not T_j implied by F_j, add this equality to F_j and go to step 2.

4: Return "Satisfiable".

share the equalities as the Nelson-Oppen method but maintains for each theory solver a candidate model of the shared equalities across the theory solvers. This candidate model propagates the equalities which provides impressive performance improvements in comparison to the Nelson-Oppen method.

5 Selected Tools

The tools we used in this work are the TreeSolver (GNU Prolog Constraint Solver 1.3.1) (61), the Choco 3 Constraint Solver library (60), Minion 0.15 CSP-solver (30), Minion 0.15 in combination with the modelling assistant Savile Row 1.0RC1 (58), Microsoft Z3 SMT-solver 4.3.0 x64 (63), and the Cooperating Validity Checker (CVC)4 SMT-solver (4). The TreeSolver was the existing CSP-solver which was used during the development phase of the tool suite STATION in the test case generation phase. This CSP-solver is restricted to process problems without using integer valuations less than zero. The Choco CSP-solver is a library for the Java programming language which is also used for development of STATION. Choco can be fully integrated in a Java project and so a lack of startup time for an external tool can be avoided. The Minion 0.15 CSP-solver is an external tool which is one of the fastest available CSP-solvers. Minion 0.15 and the combination of Minion 0.15 and Savile Row 1.0RC1 are developed at the University of St Andrews where this work was accomplished supported by the VIRTUAL VEHICLE Research Center. The Microsoft Z3 SMT-solver is an external tool which has always been among the best in the SMT competition since it emerged. The CVC4 SMT-solver is the latest of the SMT-solver- also theorem-prover-family developed in a joint project of the New York University and the University of Iowa. Both, Z3 and CVC4, support the SMT-LIB Version 2.0 (7) syntax as input, provide executables for Microsoft Windows Operating System (OS)s and include nearly all needed built in theories for this analysis.

5.1 TreeSolver

The TreeSolver uses in its core the GNU Prolog 1.3.1 CSP-solver (56). As the name indicates the core is a free CSP-solver implemented in the Prolog logic programming language. This core of the CSP-solver is an instance of the Constraint Logic Programming (CLP) scheme introduced in (33). The TreeSolver is delivered as executable for Microsoft Windows OSs. The applied version only allows usage of positive integer valuations. Another drawback of the TreeSolver was that it only provides a network socket for communication which restricts the length of the input.

5.2 Choco

The Choco 3 CSP-solver library is a completely rewritten successor of the Choco 2 version. At the time when this work was prepared the used Choco 3 release was still in a beta state. Choco is implemented in Java and deployed as a library which makes it easy to be integrated in a Java development project what this work‘s basic development language is.

5.3 Minion

The Minion CSP-solver is a stand-alone CSP-solver developed at the University of St Andrews where this work was realized. Minion is very fast and scales very well as the problem size increases. Minion processes models created in its own syntax delivered as input text file or via input stream. In this work we used the 32-bit version which is available to download as executable for Microsoft Windows OS and a 64-bit version which is not officially available as executeable for Microsoft Windows OS. In the remainder of this work we label the 32-bit version of Minion as Minion32, the 64-bit version of Minion as Minion64, and if not differentiating we use simply Minion.

5.3.1 Savile Row

Savile Row is a modeling assistant for Constraint Programming (CP) which is a stand-alone application developed at the University of St Andrews. Savile Row processes input files in the Essence’ modeling language which is a high-level language for users to specify a CSP and translates the input to the input of a CSP-solver. The translation applies some reformulations to improve the Essence’ input. The CSP-solver used in this work in combination with Savile Row was Minion32 where Savile Row exports a file which is a proper input file for the Minion CSP-solver.

5.4 Microsoft Z3

Z3 4.3.0 is a stand-alone SMT-solver developed at Microsoft Research and is available as open source. Z3 supports processing of inputs written in different syntax formats. The format chosen in this work was the SMT-LIB Version 2.0 syntax which is an actual standard and is supported by several SMT-solvers. The input for Z3 is delivered in

a text file. Z3 is available as executable for Microsoft Windows OS in a 32-bit and a 64-bit version. In this work we chose the 64-bit version.

5.5 CVC4

CVC4 1.2 is the latest version of a stand-alone SMT-solver developed in a joint project of the New York University and the University of Iowa and is available as open source. It supports processing of inputs written in the SMT-LIB Version 2.0 syntax. It is a completely rewritten successor of CVC3. CVC4 is available as executable for Microsoft Windows OS in a 32-bit version.

6 Test Input

For testing input formulas in FOL were generated using variables of type integer and formulas in propositional logic were generated using variables of type boolean. These formulas represent a constraint path in an ESTS. To get a meaningful set of test inputs to analyze the differences of the applied CSP-solvers and SMT-solvers, formulas of different length using the currently supported operators and data types of ESTSs were generated.

6.1 General Test Structure

The structure of test inputs using boolean variables is a formula in propositional logic consisting of nested subformulas which are connected by a logical `AND`-operator or a logical `OR`-operator, acting as logical connective, listed in Table 6.1. Subformulas again can have subformulas. These subformulas have a simple structure where each subformula starts with either another subformula, a variable, or a boolean value $b \in \{True, False\}$ followed by a logical connective, repeats this as often as the set length admits, and ends either with a subformula, a variable, or a value b. The logic NOT operator from Table 6.1 is randomly applied to variables, subformulas, or the entire generated formula. The length is a parameter set by the user, who generates the test inputs.

The test inputs using integer variables we used to run on the different solvers are quantifier free FOL formulas. The structures and semantics of FOL are explained in detail in (31, 13). A term t used in these FOL formulas is either a constant, a variable, or a binary function which again take terms as arguments. This term t evaluates to a valuation $v(t)$ where $v(t) \in \mathbb{Z}$. The binary functions used as a term t are listed in Table 6.2 which contains the arithmetic operators supported by an ESTS. A list of the predicates which are used in the FOL formulas is shown in Table 6.3. These predicates are binary relational operators and applied on terms. A predicate evaluates to a literal which is either *True* or *False*. A FOL formula representing a generated test input is a literal or the application of a logical connective from Table 6.1 to a formula or formulas.

<table>
<tr><th>name</th><th>Logic AND</th><th>Logic OR</th><th>Logic NOT</th></tr>
<tr><td>symbol</td><td>&&</td><td>||</td><td>!</td></tr>
</table>

Table 6.1: Logic operators

name	addition	subtraction	multiplication	division	modulo
symbol	$+$	$-$	$*$	$/$	%

Table 6.2: Arithmetic operators

6.2 Test Generation

To test different data types of variables we decided to split the generation of test inputs using boolean variables and the generation of test inputs using integer variables. The generation of test inputs using boolean values was led by setting the length for the generated propositional formulas where the length represents the number of logical connectives from Table 6.1. The nesting of subformulas was decided randomly. In the remainder of this work we label the set of test inputs using Boolean variables as boolean-set.

For the generation of the test input for integer variables we decided to group the generated FOL formulas by their arithmetic operators. We generated a set of FOL formulas for each operator in Table 6.2 and another set using all operators appearing randomly. In the remainder of this work we label this six sets as `+`-set, `-`-set, `*`-set, `/`-set, `%`-set, and random-set corresponding to the arithmetic operators they contain in their FOL formulas. The general grammar used for randomly created FOL formulas is represented in Listing 6.1. The selection of the operators, the construction of identifiers, the building of constants, and the generation for the non-terminals: formula, literal, predicate, function, and term was performed randomly. These non-terminals are introduced and explained in Section 6.1. Only the recursion depth for the non-terminals was set to a max. value to constrain the length of the generated FOL formulas. The used path to produce the FOL-formulas can be found in Listing 6.1 to reproduce the FOL formulas.

In Definition 1 the length of a generated FOL-formula for the different sets except the random-set is defined. The expressions used in Definition 1 correspond to the expressions used in Listing 6.1. Consider that in Definition 1 the variable `n` provides an input to define the resulting number of boolean operators within an FOL-formula created during its generation. The function `length(n)` in Equation 6.1 returns the number of used terms, the number of arithmetic operators, the number of relational operators, and the number of boolean operators of a generated formula. Since `length(n)` depends on

name	equal	not equal	greater than	less than	greater than or equal	less than or equal
symbol	==	!=	>	<	>=	<=

Table 6.3: Relational operators

the same parameter as used for the generation of the FOL-formulas, it is not applicable in general. However, it reflects the number of expressions created by the generation algorithm shown in Algorithm 4. For this reason it provides a comparable measure for the SMT- and CSP benchmark.

```
1  formula    : literal
2             | predicate
3
4  literal    : formula boolean predicate
5             | not formula boolean predicate
6
7  predicate  : function relational function
8             | not function relational function
9
10 function   : function arithmetic function
11            | term
12
13 term       : identifier
14            | constant
15
16 arithmetic : + | - | * | / | %
17 relational : == | != | > | < | >= | <=
18 boolean    : && | ||
19 not        : !
20 identifier : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|
21              'A'..'Z'|'_'|'0'..'9')*
22 constant   : ('0'..'9')+|'-'('1'..'9')('0'..'9')+
```

Listing 6.1: FOL formula symmetric arithmetic structure

Definition 1.
The function `length(n)` *returns the length of a generated formula f, where the length consists of a number of terms, arithmetic operators, relational operators, and boolean operators* n. □

$$length(n) := \begin{cases} terms & = 2(n+1)^2 \\ arithmetic\ operators & = 2(n+1)^2 - 2(n+1) \\ relational\ operators & = n+1 \\ boolean\ operators & = n \end{cases} \tag{6.1}$$

An algorithm to show the creation of the FOL-formulas for the random-set is shown in Algorithm 4. To apply this algorithm for non random structured formulas the number of desired boolean operations `n` is assigned to variable *literal_recursion* and the function GET_RANDOM_INTEGER() returns the value of *literal_recursion* after each call. The complexity of this algorithm is in the worst case $O((literal_recursion + 1) * 2 * (function_recursion + 1))$. Due to the fact that the max. value for *literal_recurion* and $function_recursion$ are the same the resulting complexity is simplified $O(max_value^2)$.

Listing 6.2 shows the grammar of a FOL formula using only the `+`-operator. The relational and boolean operators, the construction of identifiers, and selection of constants were performed randomly. The length of the generated FOL formulas is constrained with a max. value for recursion depth. This structure was used for every single arithmetic operator to generate the test inputs containing only one arithmetic operator in one test input.

To ensure differences in the structure of the generated FOL formulas we decided to lead the generation to create at the level of arithmetic operators a symmetric structure, a more degenerated structure, and a randomly mixed structure of both. The symmetric structure is obtained by running the recursive calls in line 10 of Listing 6.1 of `function` at the left and the right of `arithmetic` equally often. In contrast Listing 6.3 shows in line 10 another method which enforces a degenerated structure of the generated FOL formula. The differences to Listing 6.1 and the affected parts are highlighted in Listing 6.3. The randomly mixed structure generation decided randomly during generation which structure to use and was also able to change during the generation time. This resulted in a mix of the symmetric and the degenerated structure for one test input. To express the structures in the FOL formulas we used parentheses to enclose related operations.

```
1  formula     : literal
2              | predicate
3
4  literal     : formula boolean predicate
5              | not formula boolean predicate
6
7  predicate   : function relational function
8              | not function relational function
9
10 function    : function arithmetic function
11             | term
12
13 term        : identifier
14             | constant
15
16 arithmetic  : +
17 relational  : == | != | > | < | >= | <=
18 boolean     : && | ||
19 not         : !
20 identifier  : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|
21                'A'..'Z'|'_'|'0'..'9')*
22 constant    : ('0'..'9')+|'-'('1'..'9')('0'..'9')+
```

Listing 6.2: FOL formula using only +-operator

The differences in the structure which is either symmetric or degenerated is important for arithmetic operators like the --operator, the /-operator, and the %-operator which are not associative. In Example 7 a FOL formula is shown, which represents the symmetric structure whereas Example 8 shows a FOL formula which represents the degenerated structure. In the remainder of this work we label the FOL formulas depending on their generated structure as degenerated FOL formula, symmetric FOL formula, and random FOL formula.

```
1   formula     : connective
2               | literal
3
4   connective  : formula boolean literal
5               | not formula boolean literal
6
7   literal     : function relational function
8               | not function relational function
9
10  function    : function arithmetic term
11
12  term        : identifier
13              | constant
14
15  arithmetic  : /
16  relational  : == | != | > | < | >= | <=
17  boolean     : && | ||
18  not         : !
19  identifier  : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|
20                 'A'..'Z'|'_'|'0'..'9')*
21  constant    : ('0'..'9')+|'-'('1'..'9')('0'..'9')+
```

Listing 6.3: FOL formula using only /-operator

Example 7 (Symmetric structure of a generated FOL formula using only the --operator).
$((x_5207 - 2071) - (x_4618 - 9979) >= (x_3470 - x_7218) - (x_9032 - -7407)) ||$
$!((x_4840 - 939) - (x_3470 - x_7218)! = (x_9032 - -7407) - (x_759 - x_2398))$ □

Example 8 (Degenerated structure of a generated FOL formula using only the --operator).
$((((x_5207 - 2071) - x_4618) - 9979) >= (((x_3470 - x_7218) - x_9032) - -7407)) ||$
$!((((x_4840 - 939) - x_3470) - x_7218)! = (((x_9032 - -7407) - x_759) - x_2398))$
□

Algorithm 4 CREATE random FOL-FORMULA

Input: A random integer value in range of 0 to *literal_recursion*
Output: A FOL-formula

```
 1: function CREATE_FORMULA(literal_recursion)
 2:     literal = CREATE_PREDICATE();
 3:     if literal_recursion > 0 then
 4:         return literal += logical connective +
              CREATE_FORMULA(literal_recursion − 1);
 5:     else
 6:         return literal;
 7:     end if
 8: end function
 9:
10: function CREATE_PREDICATE
11:     return CREATE_FUNCTION(GET_RANDOM_INTEGER()) +
              relational operator +
              CREATE_FUNCTION(GET_RANDOM_INTEGER());
12: end function
13:
14: function CREATE_FUNCTION(function_recursion)
15:     function = get_variable();
16:     \\ get_variable() either returns an integer value or a variable name
17:     if function_recursion > 0 then
18:         return function += arithmetic operator +
              CREATE_FUNCTION(function_recursion − 1);
19:     else
20:         return function;
21:     end if
22: end function
23:
24: function GET_RANDOM_INTEGER
25:     return a random integer value > 0;
26: end function
```

7 Experimental Results

7.1 Environment

The results in this work were obtained by using a certain environment of hardware and software tools. The device used as hardware environment had following specifications:

- Dell Precision M4500 Mobile Workstation
- Intel Core i7 Q740 1.73 GHz Quad-Core
- 8GB RAM DDR3
- Seagate Harddisk 500GB, 7200RPM, 16MB Cache

An overview of the software tools which were used in this work aside from the CSP- and SMT-solvers is given in the following list:

- Windows 7 Professional 64-bit
- Eclipse 3.7.2 IDE
- JavaSE 6
- Dumont parser (25)
- JUnit 4.8.2
- Dataram Ramdisk 4.1
- jython 2.5.2
- jregex 1.2

7.2 Setup

The initial point of running the experiments is a set of FOL formulas as explained in Chapter 6. In total we generated a set of 4760 formulas which are distributed as follows:

+-set	900
−-set	900
*-set	648
/-set	900
%-set	890
boolean-set	180
random-set	342
total	4760

Every single formula of these 4760 formulas was preprocessed and transferred as input to the applied solvers. The preprocessing was started by bringing the formula in a tree structured object called *Simple Node* by an external parser tool called *Dumont Parser*. A *Simple Node* represents a binary tree with a constant or a variable as leaf nodes and operators as the remaining nodes. The domain for the variables in each FOL formula was set to [-10000..10000] and for the propositional logic formulas the domain of the values was set to {True, False} in this step.

The next stage contains the compilers which take a *Simple Node* as input and transfer it to a proper input of the different solvers where for each solver except the SMT-solvers a unique compiler was used. The SMT-solvers use a common compiler which translates the *Simple Node* to the SMT-LIB Version 2.0 syntax.

The input for the solvers is delivered in different ways. As Minion, Savile Row, and CVC4 read from a standard input stream, in Choco you use the API to form proper input. The GNU Prolog solver uses the TreeSolver (61) which is a simple server offering a socket connection to connect to the GNU Prolog solver. This allows to send constraints in a tree structure to the TreeSolver which replies either a solution or an error message. Z3 requires a RamDisk to circumvent weaknesses in its standard input stream parser. The tool Dataram Ramdisk 4.1 (57) provided us the necessary features to write files to a certain space in RAM to avoid delays of writing data to the hard disk and reading the data from there.

Each solver was limited with a maximum time to solve of 20 seconds. We appointed this limit after running some tests which showed up that the density of the time to solve gets very wide scattered above this limit what may affect the results of this work slightly. However, this makes a solver not more useful in terms of test case generation which should finish within a certain time.

To verify the results of the different solvers we had to parse the results and then transfered them to a Jython interpreter. With this interpreter we checked if the model is indeed a model which satisfies the FOL-formula given as input. We recorded the result which was either True for a satisfying model, or False for a not satisfying model or if the time to solve exceeded the time limit. The tests were launched within the Eclipse IDE using the JUnit library. The recorded time in the results was the time JUnit timed.

A graphical overview representing the procedure of the preprocessing, the compilation and the verification is provided in Figure 7.1.

7.3 Results and Evaluation

The results of this work show the number of satisfied FOL-formulas plus the ones which were correctly identified as unsatisfiable and the time consumption for each of the chosen solvers. In the evaluation section we discuss the main reasons which caused the differences of the obtained results.

7.3.1 Test results

The experimental results were obtained by running all the generated formulas as explained in Chapter 6 as test inputs. We grouped these results corresponding to their test set.

In the provided charts representing the results (Fig. 7.2, 7.3, 7.4) the upper limit of the time axis is 20000ms. A test run which is interrupted if the time to solve exceeds 20000ms and a test where the check if the returned model satisfies the test input fails are both represented as points at 20000ms in the charts.

The first test set we executed was the boolean-set. The results in Figure 7.2 show that the used test set which includes propositional formulas can be solved easily by each solver. An information we obtained by these results though is that we figured out the start up time for each solver. The start up time is the time to solve of the used environment without the time just the solver consumed for processing a solution. This start up time includes the preprocessing and compilation time as well as the time to verify the result. This start up time increases correspondingly to the length of the

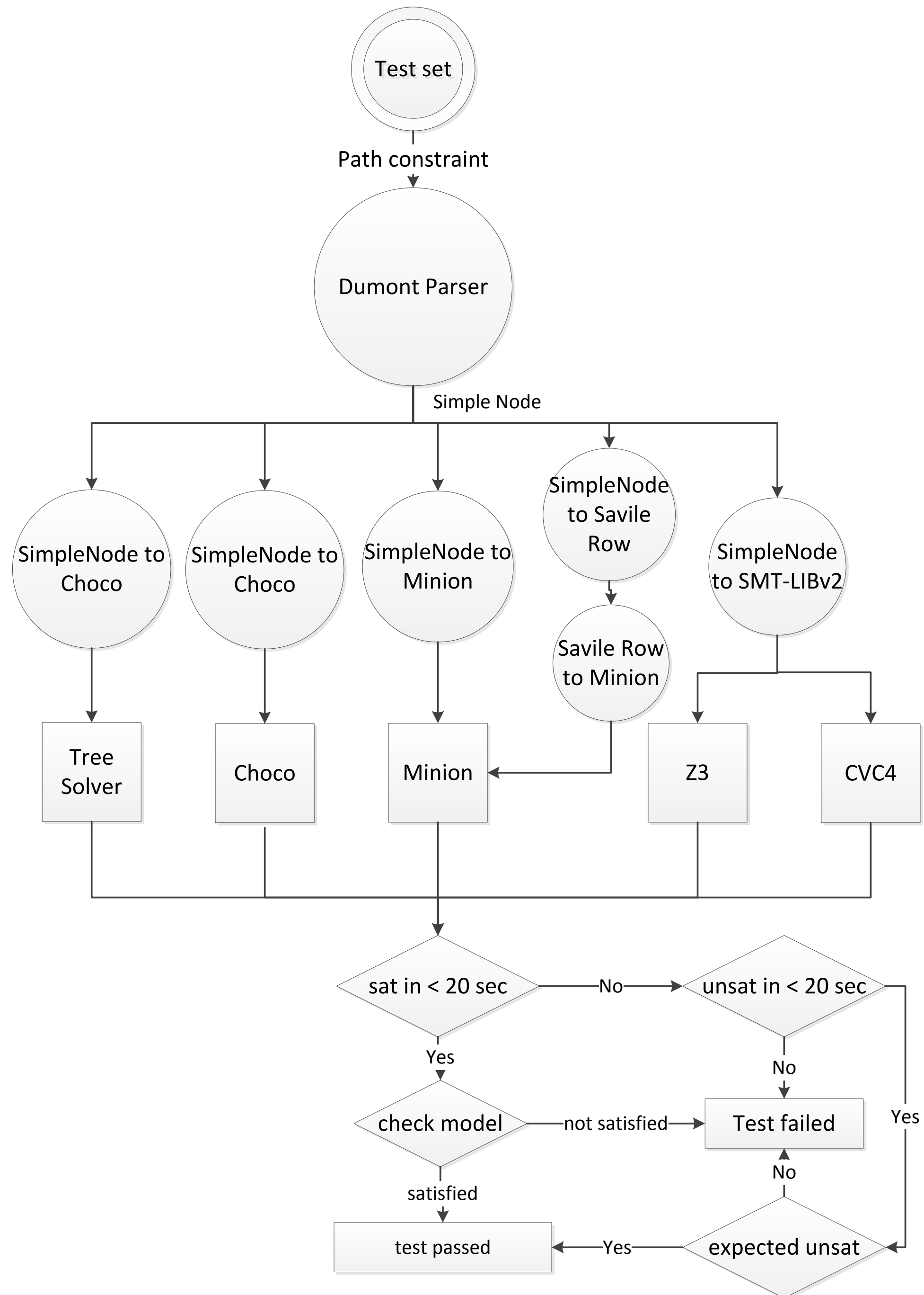

Figure 7.1: Concept of the test environment

test formula. This information can be obtained because the time which just the solver consumed for processing a solution is very small for these test formulas. Except from the TreeSolver all solvers finish all tests correctly.

Further results were obtained by running the +-set. This set is considerably larger than the boolean-set and it contains also some easier tests which do not pose a challenge for the solvers and finish as fast as the tests in the boolean-set. As stated in Chapter 6 the formulas increase in length and therefore in complexity and so there are tests in this set which none of the solvers is able to solve within the provided time. Most of the tests (95.7%) are satisfied correctly by the SMT-solver Z3 followed by CVC4, then the both versions of Minion, then Savile Row with Minion, then Choco and at last the TreeSolver. The results for the +-set are shown as a chart in Figure 7.3.

The results for the --set are very similar to the results of the +-set with the same order of the solvers corresponding to the satisfied tests. For the *-set Minion64 satisfied more tests than CVC4 and Minion32 but still less than Z3. The test inputs from the *-set take more time to solve than the tests for the +-set and the --set.
Moreover the results for the /-set show that Z3 is again the solver which satisfies most formulas of all solvers. The differences to the other solvers are rather big in the results of the /-set in comparison to the previous sets. Z3 is followed by Minion64, then Minion32, Savile Row, CVC4, the TreeSolver, and at last Choco which only satisfies 1.6% of the tests of the /-set.
The results obtained by the %-set are similar to those obtained by the /-set.

Finally we obtained the results for the random-set. These results provide a reflection of the already obtained results where we figured out that Choco can be very fast but does not satisfy a lot of formulas in comparison to other solvers. Further we figured out that the TreeSolver satisfies only a small number of formulas within the 20 seconds time limit. Again for the random-set the Z3 solver satisfies the most formulas followed by Minion64, then Minion32, Savile Row with Minion32, CVC4, Choco, and the TreeSolver. The results for the random-set are shown in Figure 7.4.

The overall results are shown in Table 7.1. The table shows the results for each single test set as well as the overall result. It shows the time to solve for each solver which it consumed to run the depicted test set, the number of passed tests, and the percentage of how many tests passed of all given tests. The charts (Fig. 7.2, 7.3, 7.4) representing the results for each test set do not differentiate a failing result where the time to solve either exceeded 20 seconds or the check of the returned model failed. In Table 7.1 the time needed to solve the CSPs is shown. The results shown at the bottom right of this table show that Z3 is not only the solver which passed the most tests it also is

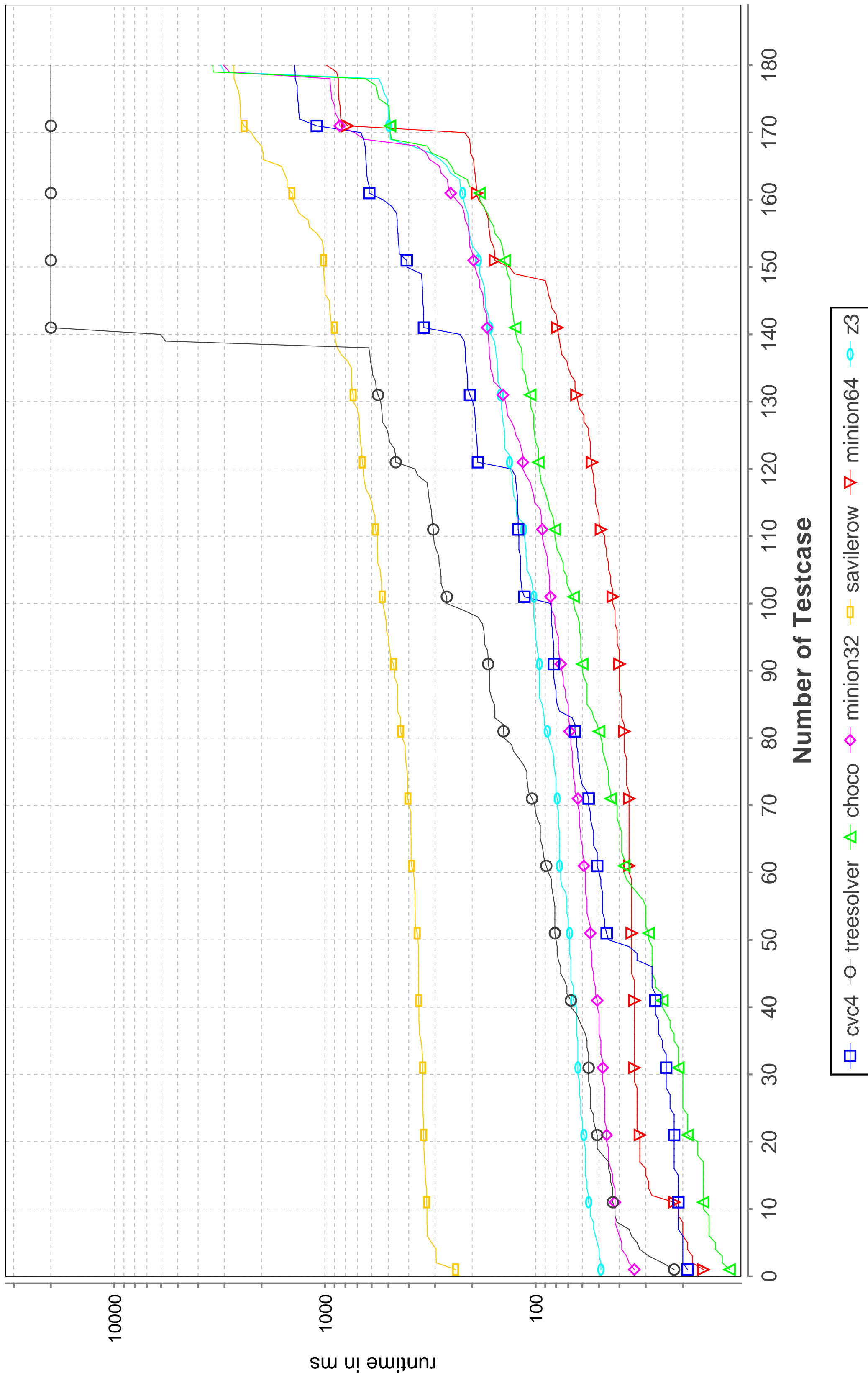

Figure 7.2: Results of running tests from the `boolean`-set

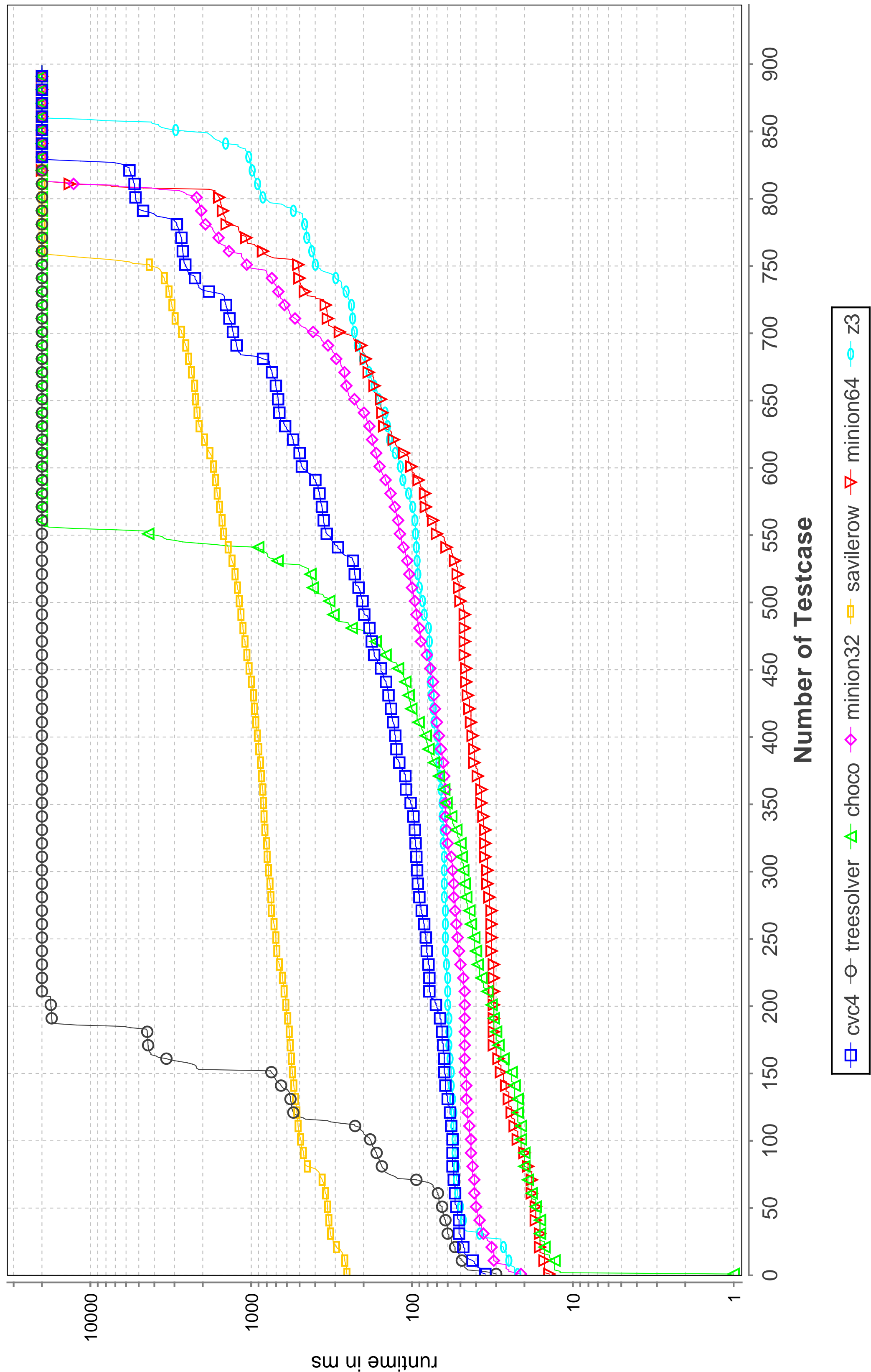

Figure 7.3: Results of running tests from the +-set

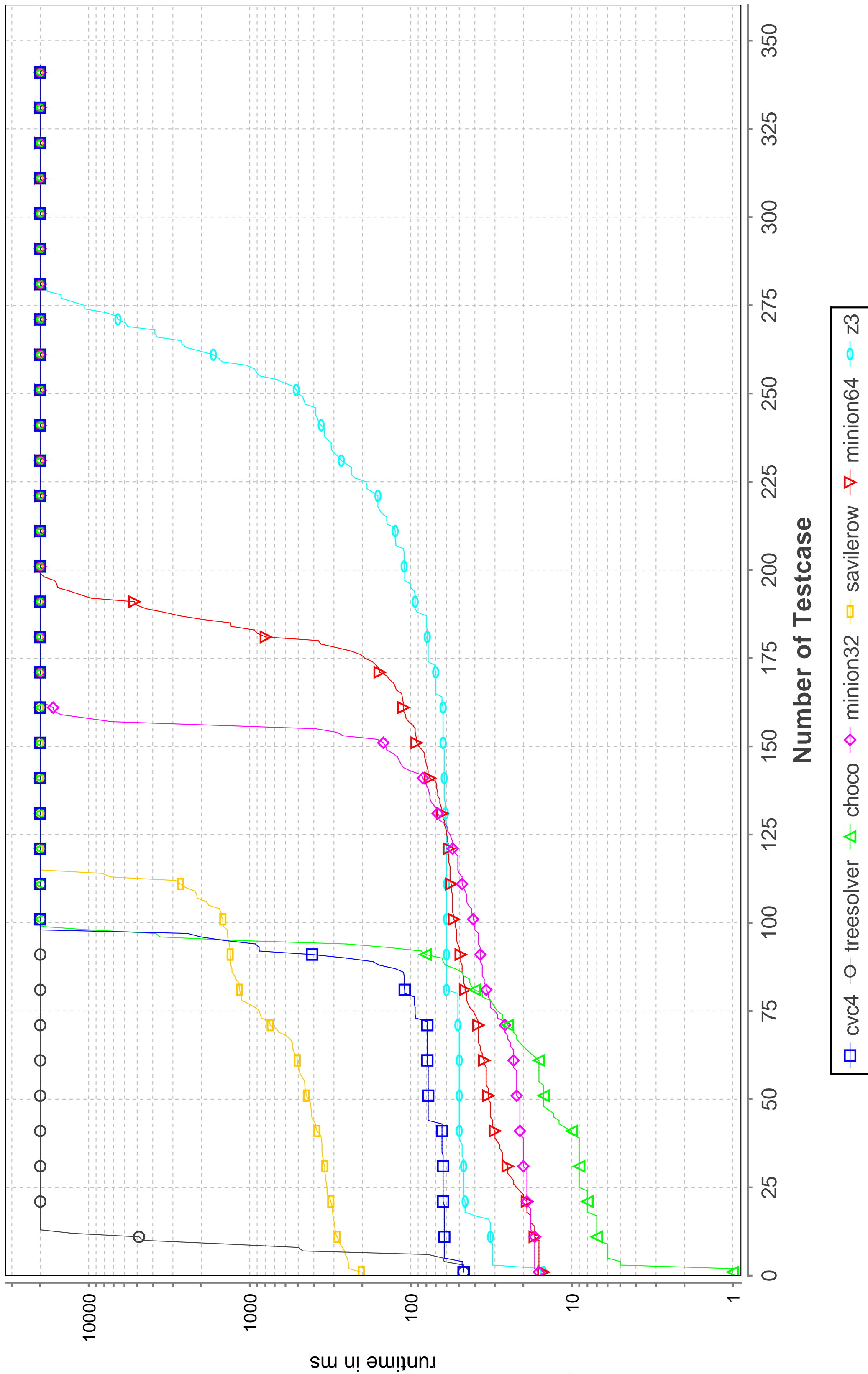

Figure 7.4: Results of running tests from the `random`-set

	boolean-set 180 formulas			+-set 900 formulas			--set 900 formulas			*-set 648 formulas		
	time	pass	%	time	pass	%	time	pass	%	time	pass	%
Z3	30.4	180	100.0	383.4	861	95.7	321.9	861	95.7	1716.0	569	87.8
Minion64	19.6	180	100.0	1803.5	813	90.3	1882.6	811	90.1	3216.3	459	70.8
Minion32	31.9	180	100.0	1872.2	813	90.3	1921.9	811	90.1	746.6	341	52.6
CVC4	39.3	180	100.0	1377.9	830	92.2	1294.7	827	91.9	638.6	418	64.5
Savile Row	131.0	180	100.0	3705.8	759	84.3	3649.3	760	84.4	8684.0	204	31.5
Choco3	25.0	180	100.0	6542.2	557	61.9	6426.5	570	63.3	3896.2	340	52.5
TreeSolver	101.9	140	77.8	1239.9	208	23.1	1040.5	213	23.7	2724.0	105	16.2

	/-set 900 formulas			%-set 890 formulas			random-set 342 formulas			all 4760 formulas		
	time	pass	%	time	pass	%	time	pass	%	time	pass	%
Z3	3363.0	673	74.8	2240.3	667	74.9	801.4	279	81.6	8856.4	4090	85.9
Minion64	5813.6	565	62.8	6807.2	532	59.8	2884.9	197	57.6	22425.0	3557	74.7
Minion32	6807.7	522	58.0	9029.3	398	44.7	2238.4	161	47.1	22648.0	3226	67.8
CVC4	1441.3	201	22.3	1198.7	314	35.3	627.0	96	28.1	6617.5	2866	60.2
Savile Row	6373.8	314	34.9	10002.3	450	50.6	4043.0	113	33.0	36589.2	2780	58.4
Choco3	15102.7	14	1.6	14745.7	78	8.8	3108.6	97	28.4	49846.9	1836	38.6
TreeSolver	3462.1	141	15.7	628.8	83	9.3	1097.2	11	3.2	10294.4	901	18.9

Table 7.1: Results for the test sets featuring time and number of passed tests

the second fastest of all applied solvers. CVC4 was the fasted solver which was more than 7 times faster than the slowest which was Choco but CVC4 is only fourth in the ranking of passed tests where Choco is sixth. Minion64 passed the second most tests which were more than Minion32 but both consumed nearly the same time to solve. The fewest of all tests with only 18.9% of 4790 tests the TreeSolver passed.

These results obtained with the +-set, --set, *-set, /-set, and the %-set provide additional information by dividing them in groups concerning their length. All of these sets contain FOL-formulas divided in 5 lengths. These lengths of the formulas in the +-set and --set are as defined in Definition 1:

- length 1 = `length(1)`
- length 2 = `length(3)`
- length 3 = `length(7)`
- length 4 = `length(15)`
- length 5 = `length(31)`

For the formulas in the `*`-set, `/`-set, and `%`-set the lengths are:

- length 1 = `length`(1)
- length 2 = `length`(2)
- length 3 = `length`(3)
- length 4 = `length`(4)
- length 5 = `length`(5)

The length for the `+`-set and `-`-set formulas is considerably longer than the others. We decided the different lengths for the formulas due to some sample tests during the generation phase of the formulas. To generate formulas of different lengths was required because longer formulas for the `*`-set, `/`-set, and `%`-set are not solvable by any of the applied solvers within the 20 seconds limit. The formulas for the `+`-set and `-`-set are longer because they are rather easy to solve in comparison to the other sets. Therefore we decided to genereate longer formulas for the `+`-set and `-`-set to identify differences to make them harder to solve.

As visible in Figure 7.5 most of the tests in the `+`-set with length 1 are solved by Z3, CVC4, and Minio64 within the first 400ms. Solving formulas of length 2 shows a similar distribution in Figure 7.6 to Figure 1 except from Savile Row which takes longer to solve a couple of tests but the total number of solved tests by Savile Row is still high.

Figure 7.7 shows that Z3, CVC4, and Minion perform very similar even for longer input formulas whereas for the other solvers the number of solved tests decreases. In Figure 7.8 for CVC4 the number of solved tests in the first category decreases vastly. The results for longest formulas of length 5 from the `+`-set are shown in Figure 7.9. In this chart also Z3 and Minion tend to get slower. They still solve most of the tests within the first 2 seconds but not as seen before below 400ms. These resulting charts are very similar for the `-`-set.

The results with length 1 in the `*`-set are shown in 7.10. In these results the CSP-solvers Choco and Minion are slightly faster than the SMT-solvers Z3 and CVC4. In Figure 7.11 the results for formulas of length 2 show that Minion32 only solves a few tests whereas Minion64 still is the best solver even for longer formulas. In this case Minin32 is not slower than other solvers like Z3. For Z3 a trend that it needs more time to solve for these formulas is identifiable. Minion32 only solves a few tests in the first 400ms. We will explain the reasons for this behavior in Section 7.3.2.

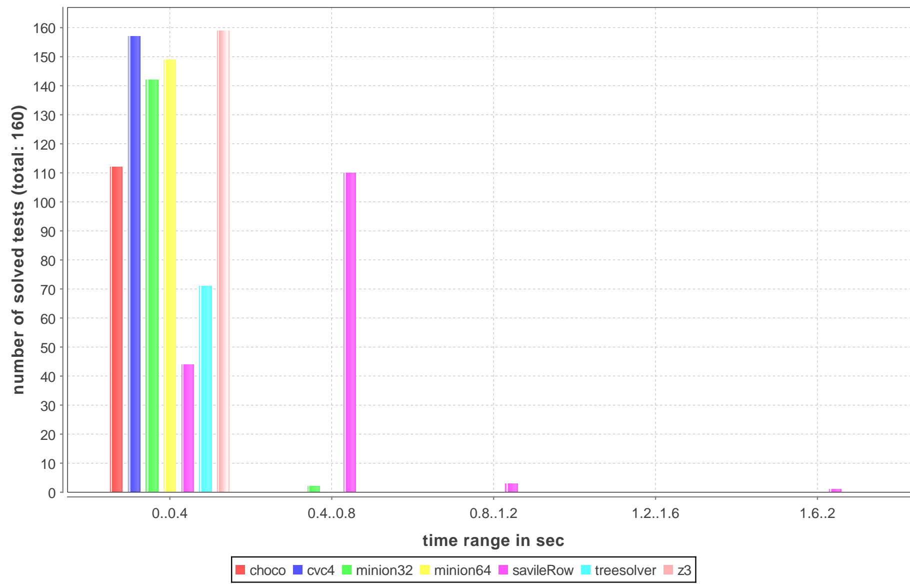

Figure 7.5: Barchart representing the number of solved tests for the '+'-set categorized in ranges of 400ms with formula length 1

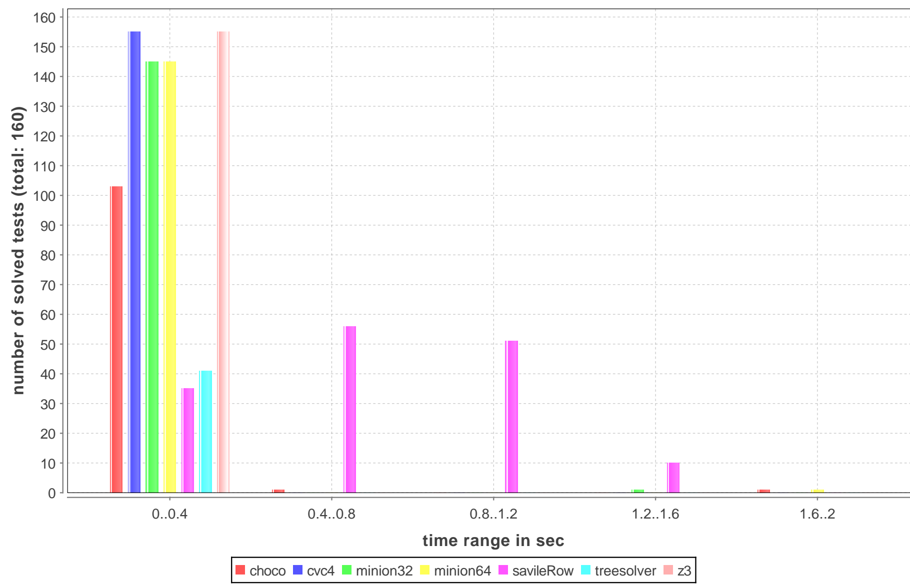

Figure 7.6: Barchart representing the number of solved tests for the '+'-set categorized in ranges of 400ms with formula length 2

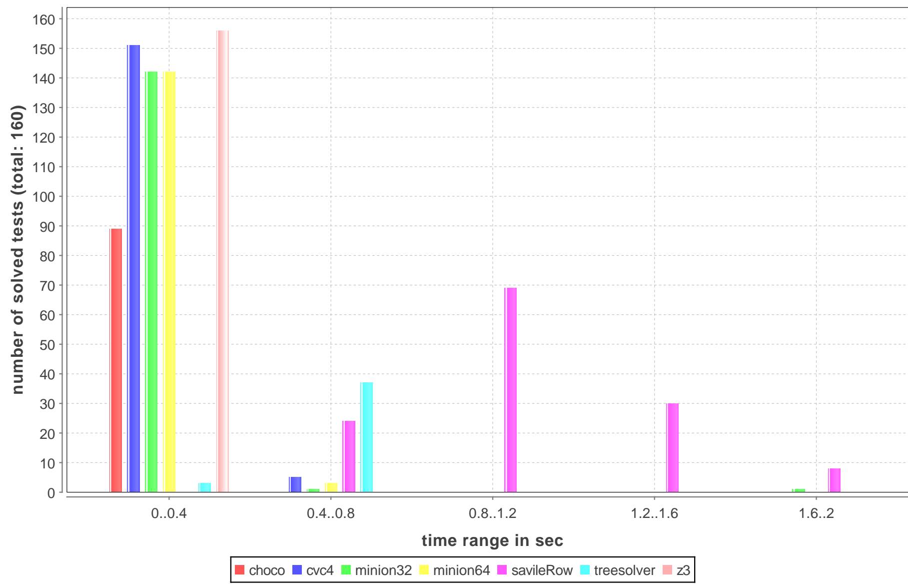

Figure 7.7: Barchart representing the number of solved tests for the '+'-set categorized in ranges of 400ms with formula length 3

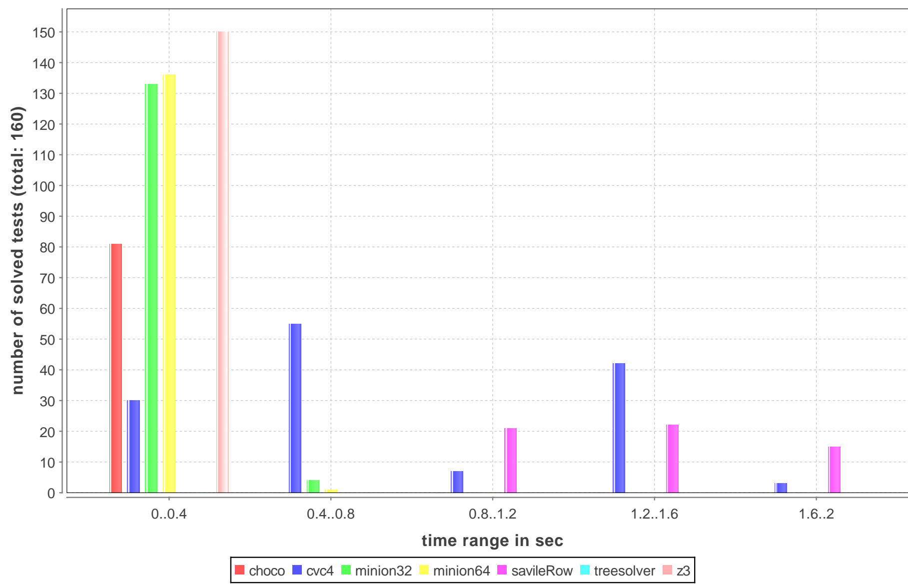

Figure 7.8: Barchart representing the number of solved tests for the '+'-set categorized in ranges of 400ms with formula length 4

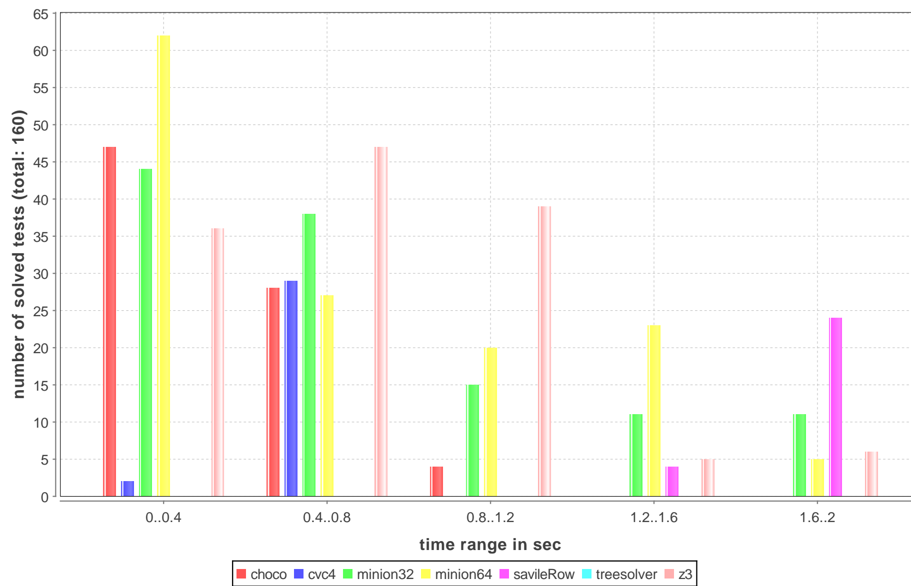

Figure 7.9: Barchart representing the number of solved tests for the '+'-set categorized in ranges of 400ms with formula length 5

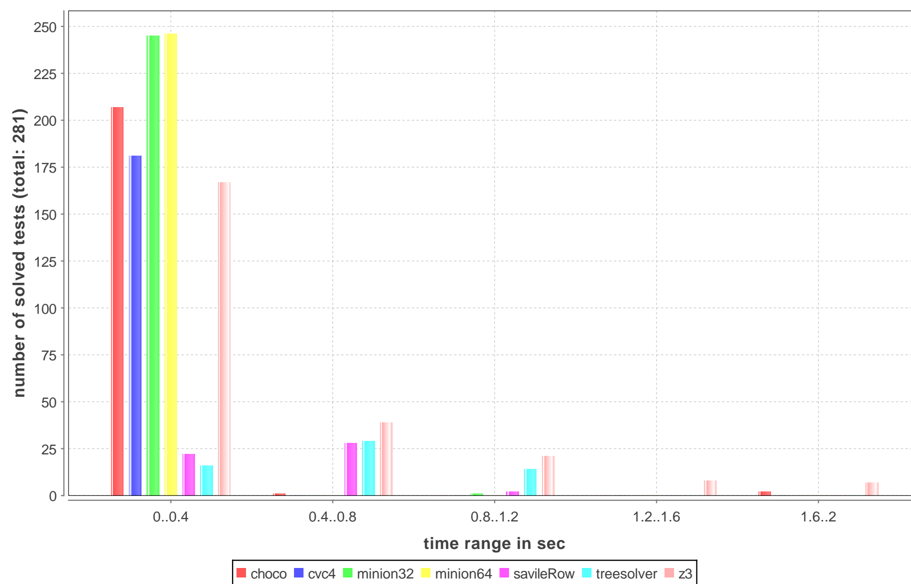

Figure 7.10: Barchart representing the number of solved tests for the '*'-set categorized in ranges of 400ms with formula length 1

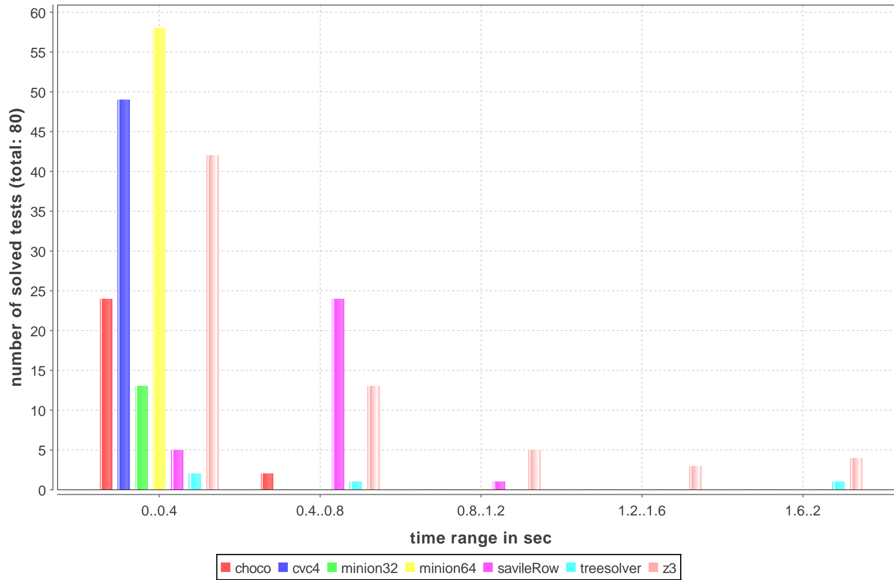

Figure 7.11: Barchart representing the number of solved tests for the '*'-set categorized in ranges of 400ms with formula length 2

In Figures 7.12 and 7.13 the number of solved tests is very similar but the SMT-solvers solve more tests than the other solvers. Figure 7.14 shows that the *-set only contains 33 formulas of length 5. From these 33 test Z3 as the best solver only solves 11 in the first 400ms.

In Figure 7.15 we can see the number of solved tests from the +-set in comparison to the total number of tests which is shown in category *total*. The other categories group the solvers and show the number of solved tests for the different lengths of formulas. This chart shows that Z3 and Minion keep very stable related to the number of solved tests if the length of the formula increases.

Similar to Figure 7.15 in Figure 7.16 the number solved tests for the −-set keeps stable for Z3 and Minion and decreases vastly for the other solvers. The small number of solved tests for the TreeSolver is caused by the formulas containing constant values less than zero or requiring a variable valuation less than zero.

For the *-set the number of generated formulas gets smaller for longer formulas. Figure 7.17 shows that Z3 solved most of the tests followed by Minion64. In these results the

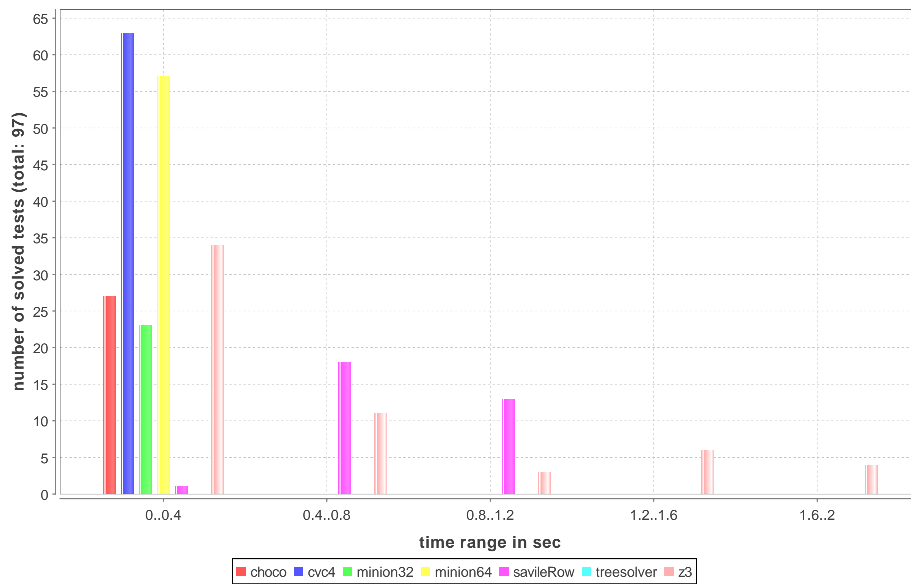

Figure 7.12: Barchart representing the number of solved tests for the '*'-set categorized in ranges of 400ms with formula length 3

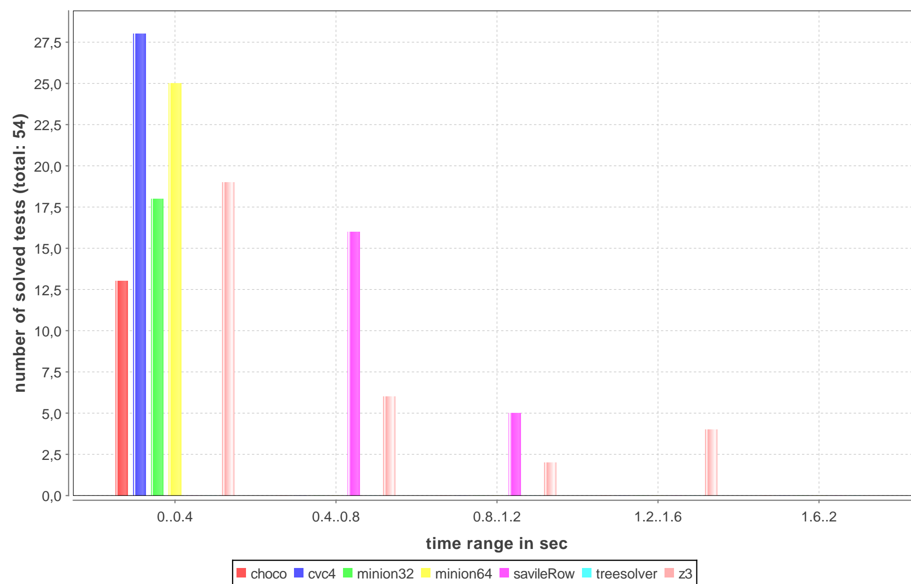

Figure 7.13: Barchart representing the number of solved tests for the '*'-set categorized in ranges of 400ms with formula length 4

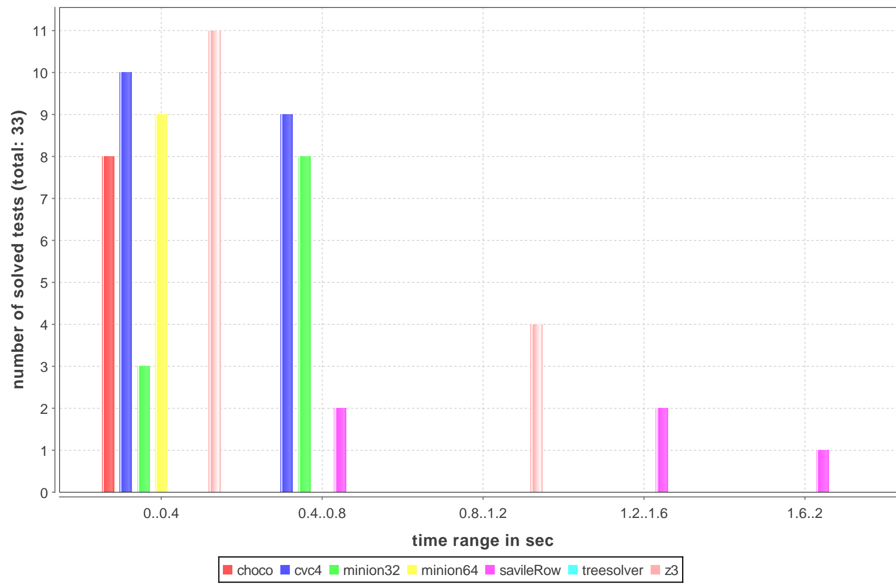

Figure 7.14: Barchart representing the number of solved tests for the '*'-set categorized in ranges of 400ms with formual length 5

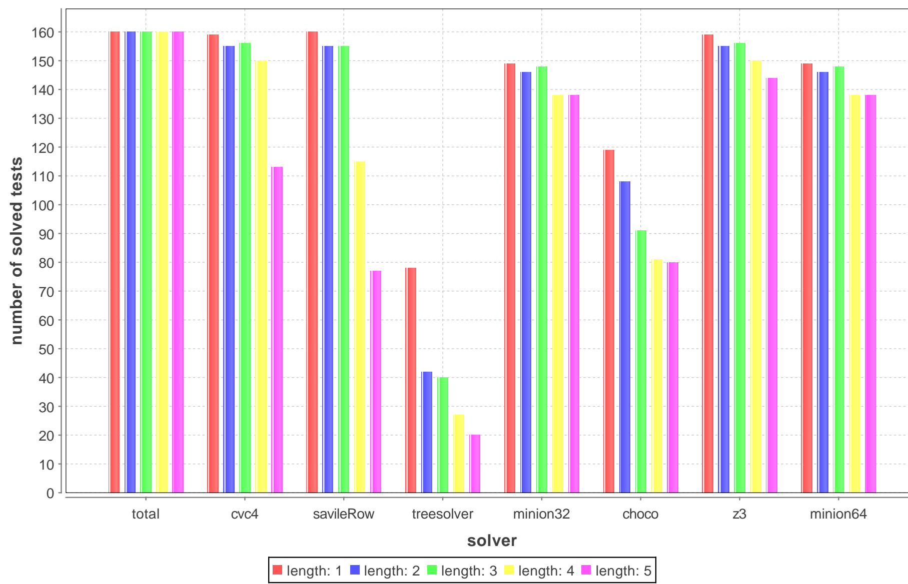

Figure 7.15: Barchart representing the number of solved tests for the '+'-set categorized by solver

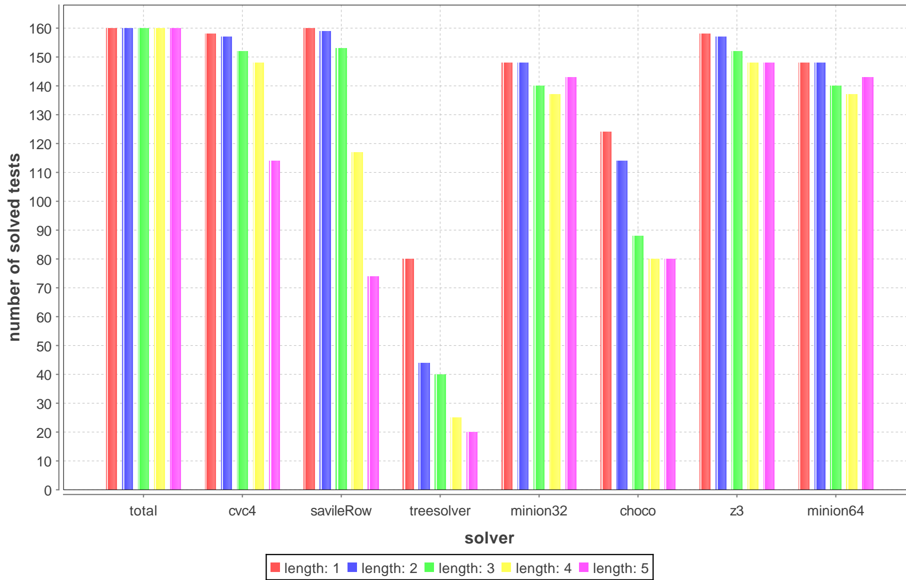

Figure 7.16: Barchart representing the number of solved tests for the '-'-set categorized by solver

number of solved tests by Savile Row for longer formulas is higher than the number of solved tests by Minion32 which uses our provided input models.

The number of tests for the different lengths varies again for the /-set. For these input formulas CVC4 performs quite weak. Minion tends to perform better for longer input formulas than Z3 for the /-set as shown in Figure 7.18.

Figure 7.19 shows the results for the %-set. These results again reveal Z3 as the best in terms of the number of solved tests followed by Minion64.

7.3.2 Evaluation

The results show considerable differences in time to solve as well as the number of passed tests. For the tests in this work the SMT-solvers were faster than all used CSP-solvers but the SMT-solvers have considerably differences in the number of passed tests as well. The CSP-solvers have huge differences in both, the time to solve and the number of passed tests.

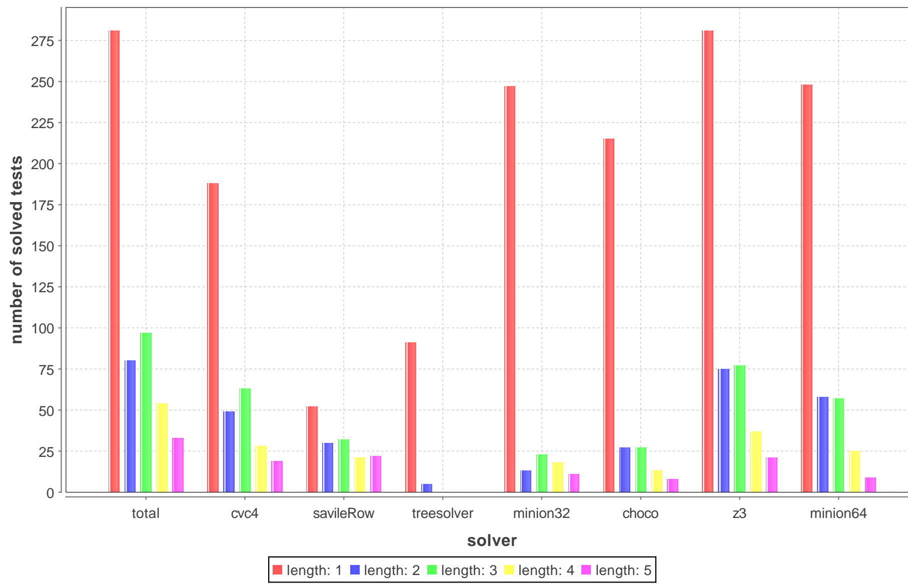

Figure 7.17: Barchart representing the number of solved tests for the '*'-set categorized by solver

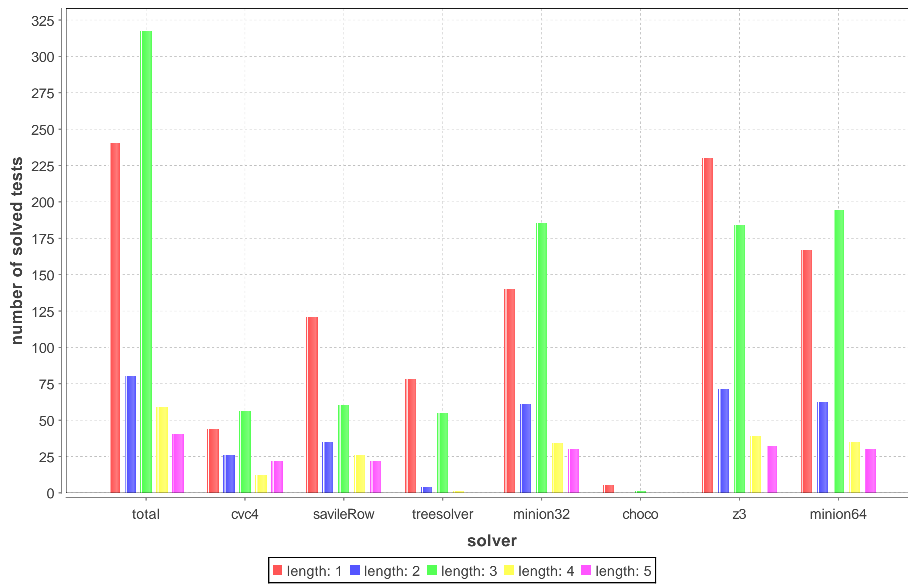

Figure 7.18: Barchart representing the number of solved tests for the '/'-set categorized by solver

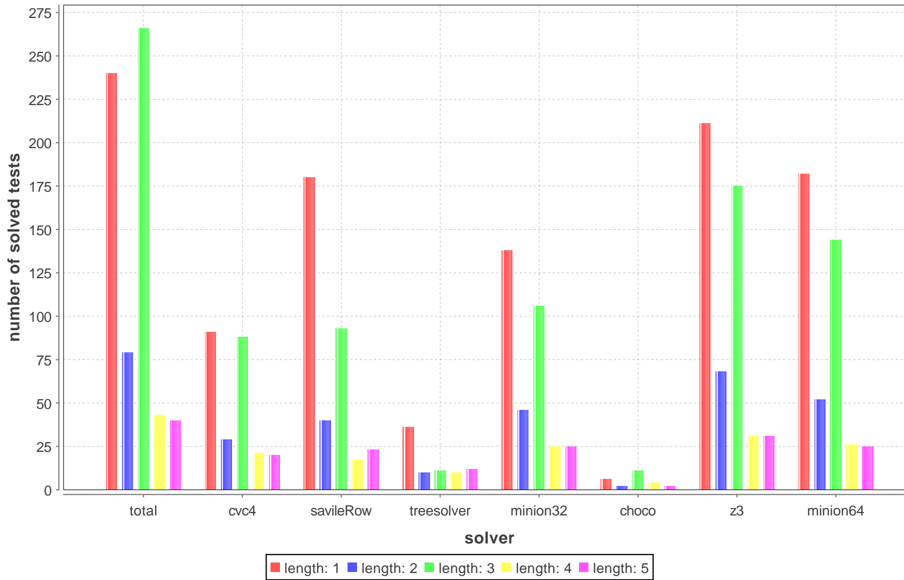

Figure 7.19: Barchart representing the number of solved tests for the '%'-set categorized by solver

The differences – as discussed in Chapter 3 and 4 – in the results of the SMT-solvers origin in the supported theories of the SMT-solver and in the heuristics used by the internal SAT-solver. As introduced in Chapter 4 the SMT-solvers use different theories related to the constraint to solve. In case of running the boolean-set both SMT-solvers consumed nearly the same time to solve which reflects mainly the startup time of the solvers. In this case the differences are hardly affected by the internal SAT-solvers or theory solvers in both SMT-solvers. These small differences are also caused by different parser implementations for the SMT-LIB Version 2.0 input and by different conversion implementations to convert the test input in CNF. The SMT-solvers CVC4 and Z3 are both based on a DPLL(T) architecture as explained in 4. Both SMT-solvers use for linear arithmetic a Simplex-based solver (23) which is integrated as theory solver in the solvers as explained in 4. The differences in the results of linear arithmetic based formulas are rather small in comparison to the results of formulas containing non-linear arithmetic. These small differences origin in different backtracking, different presimplification, and different theory propagation implementations. The differences get much clearer if formulas contain non-linear arithmetic because CVC4 in the used version does not support non-linear arithmetic neither for integer nor for reals. Z3 selects the needed theory automatically what is very handy for the user. In case of non-linear arithmetic it uses the capability of non-linear real arithmetic, which is decidable, to solve formulas in non-linear integer arithmetic. CVC4 immediately recognizes input which cannot

be solved like multiplications or divisions requiring non-linear real arithmetic and terminates delivering an error message. Due to these facts CVC4 is the fastest of the solvers as shown in 7.1 in the overall result. CVC4 does not solve formulas containing non-linear arithmetic. Because of weaker performance in the for this analysis essential theory solvers and the SAT-solver (MiniSAT (24)) as identifiable in Figure 7.2, CVC4 does not solve the same number of formulas as Z3 does.

The differences of the CSP-solvers origin in the different technologies they are based on and in their implementations. The TreeSolver uses unification and the propagation technique of arc consistency explained in Section 3.4.2 for solving constraints. The weaknesses of the TreeSolver are the support of a finite domain for variables in a max. range of [0..integer_max], where integer_max is the maximum supported positive integer value (here 2^{32}), and that the communication via the socket constrains the length of the input. The start up time of the TreeSolver which acts as a server and the arrangement of the communication are of no consequence in the results of this work. An advantage of the TreeSolver is that it answers very fast for formulas it is not able to solve.
The applied version of the Choco 3 CSP-solver was the beta version, which was available at that time. This version of Choco 3 was the only CSP-solver in this work which provided a Java API to use it as a library. Due to this fact Choco 3 has no additional start up time as the other solvers which are standalone tools have. Thus the Choco 3 CSP-solver performs very good for formulas of the boolean-set but has weaknesses in propagation of arithmetic constraints. Especially when solving formulas with solutions containing negative valuations Choco 3 caused occasionally memory overflows or did not finish within the 20 seconds time limit.
The chain using the conversion of the formula to an input for the Savile Row modelling assistant and Minion32 had the longest start up time caused by the fact that 2 standalone tools have to be started for solving a formula and an additional input delivering process from Savile Row to Minion32 was required. The advantage of this chain was that the input of Savile Row is nearly equal to the input of the Dumont parser but the boolean operators use different symbols. After substituting these boolean symbols the formula could be delivered as input to Savile Row which translates it automatically in the desired input language of the CSP-solver, in our case Minion32. During this translation Savile Row applies some reformulations to improve the model of the given input formula. One popular reformulation it does is Common Subexpression Elimination (CSE) where equally appearing expressions are replaced with a single variable everywhere it appears. This improves constraint propagation but the reformulation process to improve the model is rather time consuming in comparison to the resulting improvement of the time needed to solve for the generated formulas used in this work. This affects the results in a way that several tests do not finish within the 20 seconds

time limit because Savile Row consumes too much time translating the input formula in a proper input for Minion32.
Minion is a well established CSP-solver which has small weaknesses in propagation of arithmetic constraints but is at its core a very fast CSP-solver as shown in this empirical evaluation (37). In our results Minion was the fastest CSP-solver and passed the most tests. The Minion32 CSP-solver was slightly slower than Minion64 but passed not as many tests as Minion64 did. Apart from the extension of the supported integer range in Minion64 the differences of the two versions come from omitting a static range check for the input formula. This range check detects in Minion32 whether an integer overflow in one of the results or intermediate results might be possible. This range check uses the upper and lower limits of the domains of the variables. Because we limited even for auxiliary variables these upper and lower limits of their domain to the min. and max. values of a 32-bit integer the max. value that could ever occur is a multiplication of two 32-bit integers what results at the maximum in a 64-bit integer for an intermediate result. This check consumes time and causes the difference in the number of passing tests which led to the differences in the results of Minion32 and Minion64.

The differences of the SMT-solvers to the CSP-solvers in the results can be reduced by changing the used heuristic of the variable ordering in a test input. In this work we used a static variable ordering which was arranged by the order in the given input formula without considering any variable ordering heuristic. The SMT-solvers use internal dynamic heuristics to chose the order of variables for valuation dynamically as explained in Chapter 4. There are also several dynamic variable ordering heuristics available for CSP-solvers but they are heavily dependent on the test input. A better heuristic is a static variable ordering preprocessed by the constraints of the input formula using the width of an ordered constraint graph to get the order. To illustrate this improvement by the ordering using the width of an ordered constraint graph we picked out three examples of a CSP as shown in Listings 7.1, 7.2, and 7.3.

Listing 7.1 represents a CSP 1 where in line 1 one of the 4760 FOL formulas we generated in this work is shown. The constraints established by this formula are indicated with a `c` followed by a number and a colon in this CSP. The lines 2 and 3 in Listing 7.1 show some simplifications of the formula. If this simplifications would not be applied there were more constraints due to the negation included in the formula. The CSP 1 contains variables `x1..x5` which are expected to be valued for the result while solving this CSP and some auxiliary variables `aux1..aux3` which are not pertinent to the delivered result.

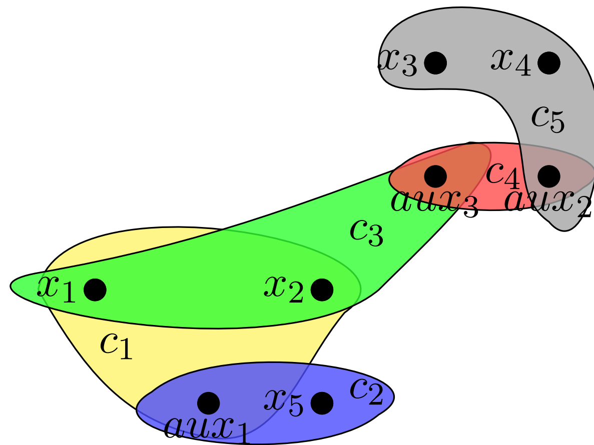

Figure 7.20: Hypergraph representing CSP 1

```
1  !(((x1 - x2) >= (x3 - x4)) || ((x1 % x2) > x5)) =
2  !((x1 - x2) >= (x3 - x4)) && !((x1 % x2) > x5) =
3  (x1 - x2) < (x3 - x4) && (x1 % x2) <= x5
4
5  c1: x1 % x2 = aux1
6  c2: aux1 <= x5
7  c3: x1 - x2 = aux3
8  c4: aux3 < aux2
9  c5: x3 - x4 = aux2
```

Listing 7.1: Constraints in CSP 1

In Figure 7.20 a hypergraph representing CSP 1 in Listing 7.1 is shown. Each subset of nodes in this hypergraph represents one of the constraints `c1..c5` from Listing 7.1. This figure lets us discern a structure in the CSP 1. Ordering the variables concerning this structure already provides a good order in sense of a fast solvable CSP. To valuate a good ordering and make the recognition of a good ordering computable we created a primal graph as shown in Figure 7.21 of CSP 1.

From this primal graph we derived an ordering using the Min-Width algorithm as shown in Listing 1. The output of this algorithm applied on CSP 1 is shown in Figure 7.22 which represents a minimum width ordered constraint graph.

As explained in Section 3.6 the width of the minimum width ordered constraint graph in Figure 7.22 is the max. number of edges leading upwards from a node in the graph. In

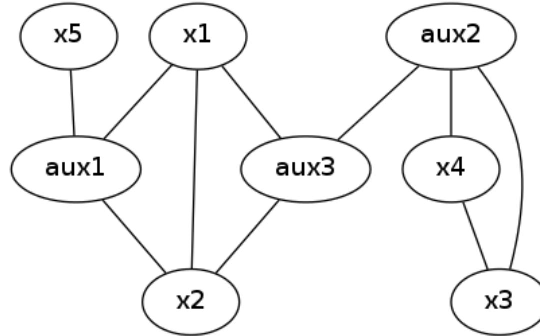

Figure 7.21: Primalgraph representing CSP 1

this graph the width is 2. This is the lowest possible width resulting from the application of the Min-Width algorithm. From this graph we derive a static order of variables `x2, x1, x4, x3, x5`. In this order the CSP-solver valuates the variables. For CSP 1 we used Minion64 to compare different orders by executing the CSP-solver with CSP 1 and all possible permutations of the variables and identified that there is no faster but several much slower orders for this CSP. The time to solve using the order derived by the minimum width ordered constraint graph was 64ms using 5 search nodes whereas in the worst case Minion64 does not solve the CSP 1 within 20 minutes and was terminated manually. A variable ordering showing the worst case is e.g. `x5,x4,x3,x1,x2`.

Listing 7.2 shows a CSP 2 established from the FOL formula in line 1. This CSP contains 5 variables `x1..x5` relevant for the result and 5 auxiliary variables `aux1..aux5`. Please note that `>=` indicates the greater or equal symbol whereas `=>` indicates an implication which was used in the generation phase of the input models of the CSP-solvers and SMT-solvers to handle the hierachical structured input from the *Simple Node*.

```
1 ((x1 >= x2 - 5840) && (x3 - -4163 == x4 * x5))
2
3 c1: aux2 <= x1 => aux1
4 c2: x2 - 5840 = aux2
5 c3: aux1 && aux3
6 c4: aux4 = aux5 => aux3
7 c5: x3 - 4163 = aux4
8 c6: x4 * x5 = aux5
```

Listing 7.2: Constraints in CSP 2

This CSP 2 from Listing 7.2 is represented as a hypergraph in Figure 7.23. This hypergraph does not let us discern a structure as clear as the hypergraph in Figure 7.20 but

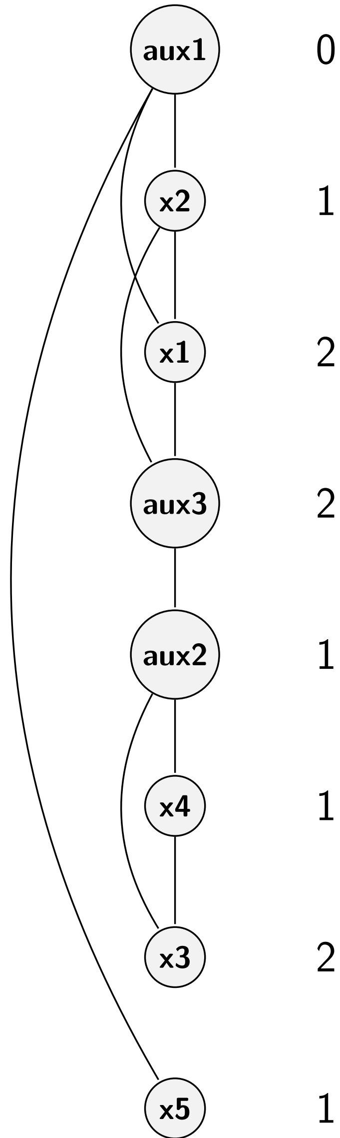

Figure 7.22: Minimum width ordered constraint graph representing CSP 1

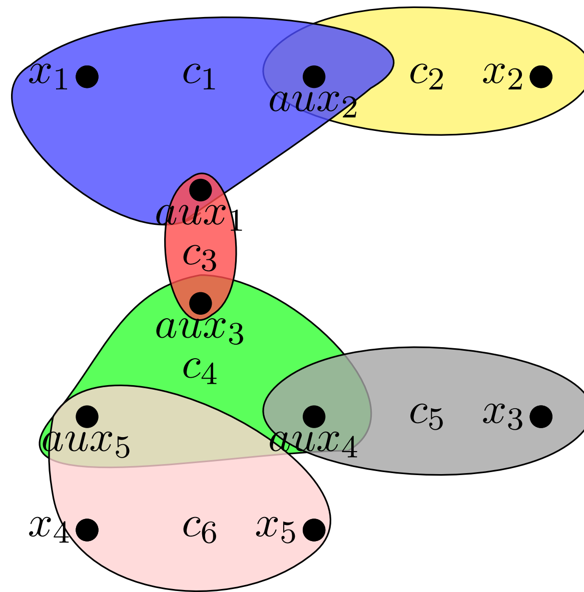

Figure 7.23: Hypergraph representing CSP 2

provides an overview of the structure of the CSP.

To show the process of creating a minimum width ordered constraint graph of CSP 2 we created a primalgraph which is shown in Figure 7.24. A minimum width ordered constraint graph compatible with this primalgraph is shown in Figure 7.25. The width of this constraint graph is 2. From this graph we derived a static variable ordering `x5, x4, x1, x3, x2` for the input of Minion64. Using this order the CSP could be solved within 58ms. Running Minion64 with the test input of CSP 2 with all possible permutations of the variable orderings yielded several orderings. These orderings were solved in a similar time as the one with the ordering we found from the minimum width ordered constraint graph. Also several orderings which did not finish within 20 minutes were found. These not finishing orderings were terminated manually (e.g. `x4, x3, x1, x2, x5`).

Another example to show the impact of a static variable ordering on the time to solve for Minion is shown in Listing 7.3. The CSP 3 in this listing contains the relevant variables `x1..x4` and four auxiliary variables `aux1..aux4`. A hypergraph of CSP 3 is shown in Figure 7.26 and represents the structure of the CSP. The CSP 3 illustrated as a primalgraph is shown in Figure 7.27. From this primalgraph a derived minimum

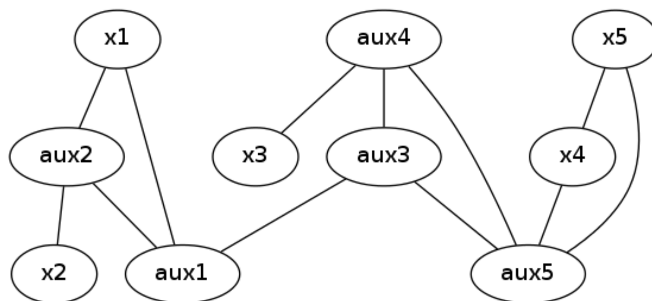

Figure 7.24: Primalgraph representing CSP 2

width ordered constraint graph is shown in Figure 7.28. This minimum width ordered constraint graph has a width of 2.

```
1 -8803 == (x1 + x2 * x3 - x4)
2
3 c1: x2 * x3 = aux4
4 c2: aux4 - x4 = aux3
5 c3: x1 + aux3 = aux2
6 c4: aux2 = -8803 => aux1
```

Listing 7.3: Constraints in CSP 3

From this minimum width ordered constraint graph we extracted a static variable ordering for CSP 3 of `x2, x3, x4, x1`. With this ordering Minion64 takes 67ms as time to solve CSP 3. Running all possible permutations yields several bad variable orderings which take several minutes to solve CSP 3. An example for a bad ordering is `x1, x3, x4, x2`.

As shown in Section 7.3.1 the length of the input formula has impact on the time to solve but for linear arithmetic less than for non-linear arithmetic. The different solvers provide their own implementations of constraints which the user can apply. A simple example for a CSP which contains two additions is shown in Example 9. This example shows that an implementation of a constraint applicable to solve additions with an arity only allowing one addition at a time requires auxiliary variables to create a CSP which contains nested additions. The domain of the added auxiliary variable in this Example has to be set to the possible min. and max. values of the intermediate result so that adding an auxiliary variable has no bearing on the solution.

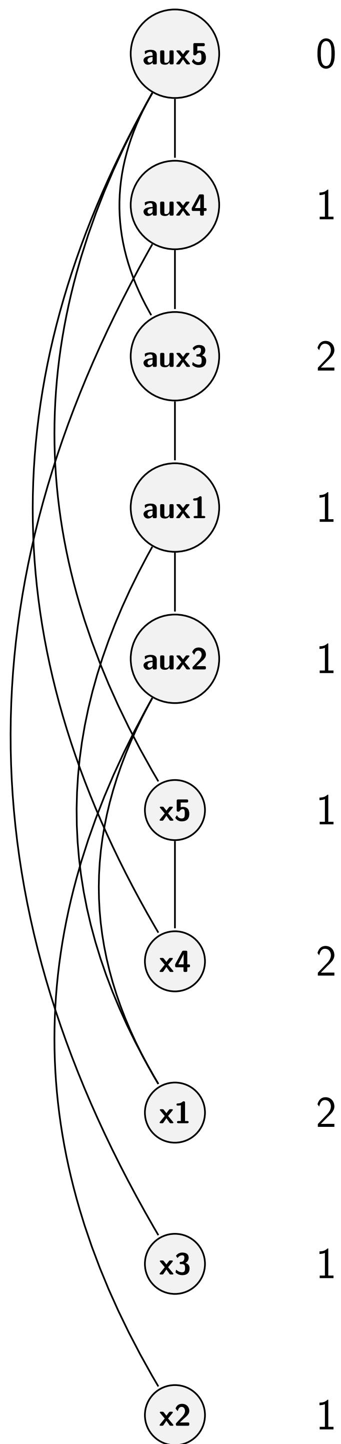

Figure 7.25: Minimum width ordered constraint graph representing CSP 2

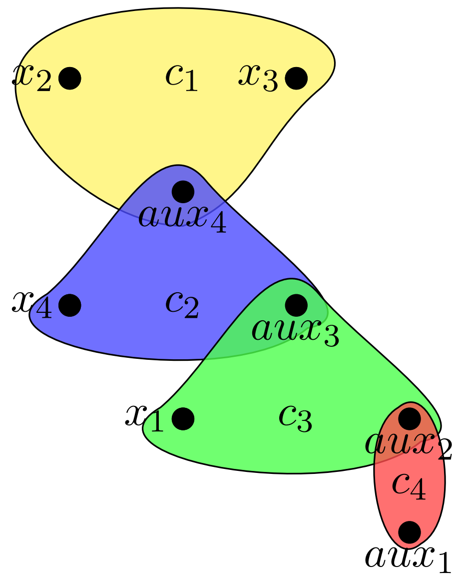

Figure 7.26: Hypergraph representing CSP 3

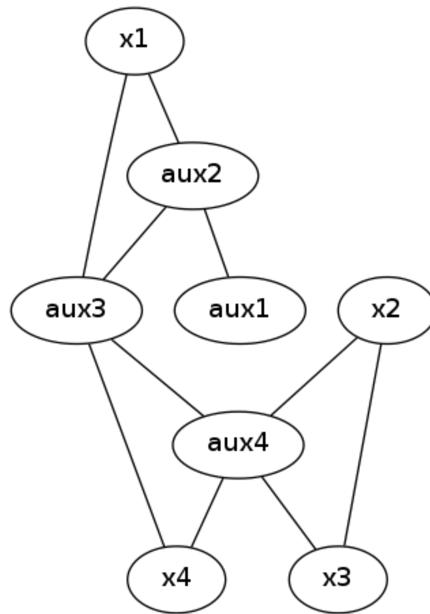

Figure 7.27: Primalgraph representing CSP 3

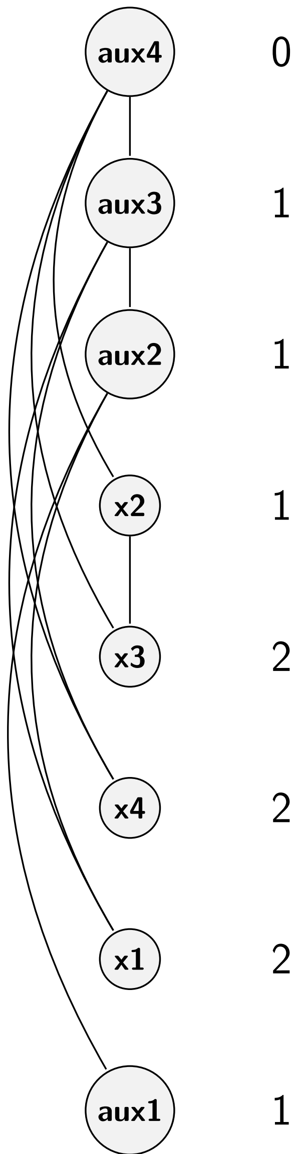

Figure 7.28: Minimum width ordered constraint graph representing CSP 3

Example 9 (Adding an auxiliary variable to create a CSP including 2 additions.).
Input formula $\mathtt{x_1 + x_2 + x_3 < 100}$ *represented as CSP:*

$\mathtt{X} = \{\mathtt{x_1, x_2, x_3, aux_1}\}$
$\mathtt{D} = \{\mathtt{D_{x_1}, D_{x_2}, D_{x_3}, D_{aux_1}}\}$ *where* $\mathtt{D_{x_1} = D_{x_2} = D_{x_3} = [-100..100]}$ *and* $\mathtt{D_{aux_1} = [-200..200]}$
$\mathtt{C} = \{\mathtt{c_1, c_2}\}$ *where* $\mathtt{c_1 := x_1 + x_2 = aux_1}$ *and* $\mathtt{c_2 := aux_1 + x_2 < 100}$ □

Adding auxiliary variables extends the search space depending on the applied operation. As shown in 9 for an addition the domain for the auxiliary variable is considerably larger than for the other decision variables. For a multiplication as shown in Example 10 the domain for auxiliary variables can grow exponentially what justifies the results for the *-set. This causes several failing tests for the solvers like Minion32 and Choco which support integer valuations in a range of $[-2^{31}..2^{31}]$. These solvers check possible overflows before processing the CSP and terminate if an overflow can occur.

Example 10 (Adding an auxiliary variable to create a CSP including 2 multiplications).
Input formula $\mathtt{x_1 * x_2 * x_3 < 100}$ *represented as CSP:*

$\mathtt{X} = \{\mathtt{x_1, x_2, x_3, aux_1}\}$
$\mathtt{D} = \{\mathtt{D_{x_1}, D_{x_2}, D_{x_3}, D_{aux_1}}\}$ *where* $\mathtt{D_{x_1} = D_{x_2} = D_{x_3} = [-100..100]}$ *and* $\mathtt{D_{aux_1} = [-10000..10000]}$
$\mathtt{C} = \{\mathtt{c_1, c_2}\}$ *where* $\mathtt{c_1 := x_1 * x_2 = aux_1}$ *and* $\mathtt{c_2 := aux_1 * x_2 < 100}$ □

The insertion of auxiliary variables and their domains has a huge impact on the time to solve. The second impact which causes longer time to solve for longer formulas is the propagation capability of the solvers. For CSP-solvers each arithmetic operation requires an individual propagation implementation to reduce the search space ideal. The SMT-solvers require a theory propagation as well. All have the applied solvers showed weaknesses for the propagation of arithmetic operations.

7.4 Discussion

The results in this work were obtained by running different CSP-solvers and SMT-solvers without any optimizations or changes in the settings. Applying the solvers in that manner yields clearly Microsoft's Z3 SMT-solver as the best choice. With the additional effort of structuring the CSPs applying the Min-Width algorithm and establishing a static variable ordering with this might enhance the results for the CSP-solvers. Further some improvements in the propagation algorithms of the constraints used in this work are required to possibly beat the Z3 SMT-solver in the number of solved tests

applied in this work whereas the TreeSolver which does not support negative integers and with its client-server architecture can not challenge the other solvers in any way.

These results were obtained with a domain of $[-10000..10000]$ for the variables in FOL-formulas applying arithmetic operations. We compared these results with some sample executions using different domain settings and obtained insights for these tests with different domain settings. This insights showed that the domain settings have significant impact on the time to solve for all solvers. The smaller the domain the shorter is the time to solve. With smaller domains also the differences between the solvers get tighter. On the other hand the time to solve for larger domains gets significantly higher. Executing a sample of FOL-formulas with different domain sizes yielded very similar results faster. Therefore we decided to run all the tests in this work with the same domain. The domain of $[-10000..10000]$ emerged as the best in terms of expressiveness and also in terms of passable time to solve.

Further we figured out that the length of the inputs we used has a greater impact on formulas using non linear operations than on linear arithmetic. As shown in Figure 7.3 the time to solve is represented by a steadily increasing line. The most impact on this increasing line has the preprocessing as shown in 7.2. For non linear operations the explosion of the search space has the most impact on the time to solve. Constraint propagation has a huge impact on the time to solve as well. An improvement of the results in this work requires a combination of a suitable variable ordering heuristic and improvements in the propagation capability of the solvers.

8 Related Work

The two groups of solvers, the CSP-solvers and the SMT-solvers, underly two different communities. Both communities have established a challenge to compare their solvers on different benchmark CSPs.

The former Constraint Solver Competition (59) which was held until 2009 was held four times and was then replaced by the MiniZinc Challenge (53). The benchmark library for the constraint challenges is published in (29). The MiniZinc Challenge requires a CSP-solver to be able to read input in MiniZinc (43) syntax. Another empirical evaluation on CSP-solvers is shown in (38) where among others also Choco and Minion were applied. In this empirical evaluation report the benchmark results for standard CSPs like the n-Queens problem are shown.
The SMT community established the SMT-COMP (6) which is a competition to compare SMT-solvers. They use benchmarks grouped by the different theories and combinations of them, because not all solvers support all theories. The benchmarks are provided by the SMT-LIB initiative (62). These benchmarks are provided in the SMT-LIB Version 2.0 syntax which is used in this work as well.
In the paper (35) the authors show experimental results on applying non-linear arithmetic benchmarks on different SMT-solvers. They also used Z3 and the predecessor of CVC4 which was CVC3 in their experiments.

In (64) the authors show an application of Minion for test case generation. They generate test inputs for different implementations which are applied to exclude a possible different behavior of the implementations using the same inputs. In the section explaining their experimental results they also mention the fact that different variable orderings have a huge impact on the time to solve or on the time to generate a test case respectively.

The SMT-solver Z3 from Microsoft was applied in their test case generator Pex which is explained in (55). Pex is a commercial tool which can be applied on implementations based on the .NET framework. The test case generation is used for monitored execution

traces to produce new test inputs which exercise different program behavior. To reason about these execution traces and to find a valuation for this execution trace the SMT-solver is required.

9 Conclusion

Model-based test case generators stand and fall with the applied CSP- or SMT-solver. The present available solvers feature differences in the support of constraints they are made for to handle as well as differences in their approaches to solve the constraints. These different approaches cause huge differences in the time to solve.

In this work we defined a set of benchmarks to compare the applied solvers. Further we discussed the achieved results and did some investigations in the causes for the differences. These investigations comprised not only the differences between the applied CSP-solvers and the SMT-solvers. Also the differences of the CSP-solvers to the SMT-solvers were analyzed.

These results and the investigations of the differences provide enough information to ease the decision which solver is best suitable for test case generation. However, these test cases are restricted to the structure introduced in this work.

9.1 Open Problems

The application of the increment (++) and decrement ($--$) operators be it pre- or post-increment or -decrement is not defined yet for CSPs. Using these operators requires a static variable ordering in the input of the solvers which is based on the occurrence of the variables in the CSP. Another approach might be the insertion of additional constraints. Both of this ideas require auxiliary variables in the input of the solvers for the variable applied in the increment or decrement operation. These auxiliary variables have to point to the proper appearance of the original variable in the result. This states an important open problem for test case generation.

9.2 Future Work

The achieved results showed rather clear differences especially in time to solve. To prove our results another test with the application of the Min-Width algorithm for variable ordering for the best CSP-solver might be interesting.

For test case generation the support of real values by the solver provides a considerable improvement of the scope of usage. Just a handful CSP- and SMT-solvers currently support real values. An analysis of the applicability of these solvers for test case generation including real values could show if it is worth to extend the scope of usage of a test case generator.

Further an interesting approach might give the inclusion of constraint optimization techniques. By means of constraint optimization it is possible to guide the variable valution process. This can be used to assign values near the upper and lower boundaries of their domain to the variables. Either to assign several different values for one variable while keeping the valuation of other variables unchanged is possible with constraint optimization techniques.

Abbreviations

CSP Constraint Satisfaction Problem

SMT Satisfiability Modulo Theory

SAT Satisfiability

CP Constraint Programming

GT Generate and Test

FOL First Order Logic

STATION STAte based system Test and simulatION

OS Operating System

CVC Cooperating Validity Checker

ESTS Extended Symbolic Transition System

CNF Conjunctive Normal Form

CLP Constraint Logic Programming

CSE Common Subexpression Elimination

SUT System Under Test

MAC Maintaining Arc Consistency

DPLL Davis-Putnam-Logemann-Loveland

DLIS Dynamic Largest Individual Sum

VSIDS Variable State Independent Decaying Sum

CAD Cylindrical Algebraic Decomposition

Bibliography

[1] Arbelaez A., Hamadi Y., Sebag M., 2009: Online Heuristic Selection in Constraint Programming, international Symposium on Combinatorial Search - 2009.

[2] Armando A., Castellini C., Giunchiglia E., 2000: SAT-Based Procedures for Temporal Reasoning, in: S. Biundo, M. Fox, Hrsg., *Recent Advances in AI Planning*, Band 1809 von *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, S. 97–108.

[3] Audemard G., Bertoli P., Cimatti A., Korniłowicz A., Sebastiani R., 2002: A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions, in: A. Voronkov, Hrsg., *Automated Deduction—CADE-18*, Band 2392 von *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, S. 195–210.

[4] Barrett C., Conway C.L., others, 2011: CVC4, in: *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, Springer-Verlag, Berlin, Heidelberg, S. 171–177.

[5] Barrett C., Dill D., Stump A., 2002: Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT, in: E. Brinksma, K. Larsen, Hrsg., *Computer Aided Verification*, Band 2404 von *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, S. 236–249.

[6] Barrett C., Moura L., Stump A., 2005: SMT-COMP: Satisfiability Modulo Theories Competition, in: K. Etessami, S. Rajamani, Hrsg., *Computer Aided Verification*, Band 3576 von *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, S. 20–23.

[7] Barrett C., Stump A., others, 2010: C.: The SMT-LIB Standard: Version 2.0, Technischer Bericht.

[8] Barrett C.W., Sebastiani R., Seshia S.A., Tinelli C., 2009: Satisfiability Modulo Theories., in: A. Biere, M. Heule, H. van Maaren, T. Walsh, Hrsg., *Handbook of*

Satisfiability, Band 185 von *Frontiers in Artificial Intelligence and Applications*, IOS Press, S. 825–885.

[9] Bartak R., 1999: Constraint Programming: In Pursuit of the Holy Grail, in: *in Proceedings of WDS99 (invited lecture*, S. 555–564.

[10] Beek P.V., 2006: Backtracking Search Algorithms, in: *Handbook of Constraint Programming*, S. 85–134.

[11] Boussemart F., Hemery F., Lecoutre C., Sais L., 2004: Boosting systematic search by weighting constraints.

[12] Boyd S., Vandenberghe L., 2004: *Convex Optimization*, Cambridge University Press, New York, NY, USA.

[13] Bradley A.R., Manna Z., 2007: *The Calculus of Computation: Decision Procedures with Applications to Verification*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[14] Bryant R.E., German S., Velev M.N., 2001: Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic.

[15] Bryant R.E., Velev M.N., 2002: Boolean satisfiability with transitivity constraints, in: *ACM Trans. Comput. Logic*, 3(4), S. 604–627.

[16] Davis M., Logemann G., Loveland D., 1962: A machine program for theorem-proving, in: *Commun. ACM*, 5(7), S. 394–397.

[17] de Moura L., Bjørner N., 2008: Model-based Theory Combination, in: *Electron. Notes Theor. Comput. Sci.*, 198(2), S. 37–49.

[18] de Moura L., Bjørner N., 2008: Proofs and Refutations, and Z3, in: *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*, Band 418 von *CEUR Workshop Proceedings*, CEUR-WS.org.

[19] De Moura L., Bjørner N., 2011: Satisfiability modulo theories: introduction and applications, in: *Commun. ACM*, 54(9), S. 69–77.

[20] de Moura L., RueßH., Sorea M., 2002: Lazy Theorem Proving for Bounded Model

Checking over Infinite Domains., in: A. Voronkov, Hrsg., *CADE*, Band 2392 von *Lecture Notes in Computer Science*, Springer, S. 438–455.

[21] Dechter R., 2003: *Constraint processing.*, Elsevier Morgan Kaufmann.

[22] Dechter R., Pearl J., 1987: Network-based heuristics for constraint-satisfaction problems, in: *Artif. Intell.*, 34(1), S. 1–38.

[23] Dutertre B., de Moura L., 2006: A fast linear-arithmetic solver for DPLL(T), in: *Proceedings of the 18th international conference on Computer Aided Verification*, CAV'06, Springer-Verlag, Berlin, Heidelberg, S. 81–94.

[24] Eén N., Sörensson N., 2004: An Extensible SAT-solver, in: E. Giunchiglia, A. Tacchella, Hrsg., *Theory and Applications of Satisfiability Testing*, Band 2919 von *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, S. 502–518.

[25] Frantzen L., 2007: *STSimulator V271007, A Library to Simulate Symbolic Transition Systems.*

[26] Freuder E., MacKworth A., 2006: *Chapter 2 Constraint satisfaction: An emerging paradigm*, in: Band 2 von *Foundations of Artificial Intelligence*, Elsevier, S. 13–27.

[27] Freuder E.C., 1982: A sufficient condition for backtrack-free search, in: *Journal of the ACM (JACM)*, 29(1), S. 24–32.

[28] Gaschnig J., 1977: A general backtrack algorithm that eliminates most redundant tests, in: *Proceedings of the 5th international joint conference on Artificial intelligence - Volume 1*, IJCAI'77, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, S. 457–457.

[29] Gent I., Walsh T., 1999: CSPLib: a benchmark library for constraints, Technischer Bericht, Technical report APES-09-1999, available from http://csplib.cs.strath.ac.uk/. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).

[30] Gent I.P., Jefferson C., Miguel I., 2006: Minion: A fast scalable constraint solver, in: *In: Proceedings of ECAI 2006, Riva del Garda*, IOS Press, S. 98–102.

[31] Harrison J., 2009: *Handbook of Practical Logic and Automated Reasoning*, 1st, Cambridge University Press, New York, NY, USA.

[32] Hentenryck P.V., 1991: Constraint Logic Programming, Technischer Bericht CS-91-05, Department of Computer Science, Brown University, Providence, Rhode Island 02912.

[33] Jaffar J., Lassez J.L., 1987: Constraint logic programming, in: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, ACM, New York, NY, USA, S. 111–119.

[34] Jovanović D., de Moura L., 2011: Cutting to the Chase solving linear integer arithmetic, in: *Proceedings of the 23rd international conference on Automated deduction*, CADE'11, Springer-Verlag, Berlin, Heidelberg, S. 338–353.

[35] Jovanović D., de Moura L., 2012: Solving non-linear arithmetic, in: *Proceedings of the 6th international joint conference on Automated Reasoning*, IJCAR'12, Springer-Verlag, Berlin, Heidelberg, S. 339–354.

[36] Knuth D.E., 1974: Estimating the efficiency of backtrack programs., Technischer Bericht, Stanford, CA, USA.

[37] Kotthoff L., 2010: Constraint solvers: An empirical evaluation of design decisions, in: *CoRR*, abs/1002.0134.

[38] Kotthoff L., 2010: Constraint solvers: An empirical evaluation of design decisions, in: *CoRR*, abs/1002.0134.

[39] Kroening D., Strichman O., 2008: *Decision Procedures: An Algorithmic Point of View*, 1, Springer Publishing Company, Incorporated.

[40] Mackworth A.K., 1977: Consistency in networks of relations, in: *Artificial Intelligence*, 8(1), S. 99 – 118.

[41] Miguel I., 2012: Constraint Programming, lecture notes.

[42] Nelson G., Oppen D.C., 1979: Simplification by Cooperating Decision Procedures, in: *ACM Trans. Program. Lang. Syst.*, 1(2), S. 245–257.

[43] Nethercote N., Stuckey P., others, 2007: MiniZinc: Towards a Standard CP Modelling Language, in: C. Bessière, Hrsg., *Principles and Practice of Constraint Programming – CP 2007*, Band 4741 von *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, S. 529–543.

[44] Nieuwenhuis R., Oliveras A., Tinelli C., 2006: Solving SAT and SAT Modulo

Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T), in: *J. ACM*, 53(6), S. 937–977.

[45] Pnueli A., Rodeh Y., Shtrichman O., Siegel M., 1999: Deciding Equality Formulas by Small Domains Instantiations, in: N. Halbwachs, D. Peled, Hrsg., *Computer Aided Verification*, Band 1633 von *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, S. 455–469.

[46] Prosser P., 1995: Forward Checking with Backmarking, in: *Constraint Processing, Selected Papers*, Springer-Verlag, London, UK, UK, S. 185–204.

[47] Robinson J.A., 1965: A Machine-Oriented Logic Based on the Resolution Principle, in: *J. ACM*, 12(1), S. 23–41.

[48] Salido M.A., 2008: A non-binary constraint ordering heuristic for constraint satisfaction problems, in: *Applied Mathematics and Computation*, 198(1), S. 280 – 295.

[49] Schwarzl C., Aichernig B.K., Wotawa F., 2011: Compositional random testing using extended symbolic transition systems, in: *Proceedings of the 23rd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'11, Springer-Verlag, Berlin, Heidelberg, S. 179–194.

[50] Seshia S.A., Lahiri S.K., Bryant R.E., 2003: A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions, in: *In Proc. DAC'03*, S. 425–430.

[51] Sterling L., Shapiro E., 1986: *The art of Prolog: advanced programming techniques*, MIT Press, Cambridge, MA, USA.

[52] Strichman O., Seshia S., Bryant R., 2002: Deciding Separation Formulas with SAT, in: E. Brinksma, K. Larsen, Hrsg., *Computer Aided Verification*, Band 2404 von *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, S. 209–222.

[53] Stuckey P.J., Becket R., Fischer J., 2010: Philosophy of the MiniZinc challenge, in: *Constraints*, 15(3), S. 307–316.

[54] Tack G., 2009: *Constraint Propagation – Models, Techniques, Implementation*, Doctoral dissertation, Saarland University.

[55] Tillmann N., De Halleux J., 2008: Pex: white box test generation for .NET, in: *Pro-*

ceedings of the 2nd international conference on Tests and proofs, TAP'08, Springer-Verlag, Berlin, Heidelberg, S. 134–153.

[56] http://gprolog.univ-paris1.fr/, 2013: The GNU Prolog web site.

[57] http://memory.dataram.com/products-and-services/software/ramdisk, 2013: DATARAM RAMDisk.

[58] http://savilerow.cs.st-andrews.ac.uk/, 2013: Savile Row.

[59] http://www.cril.univ-artois.fr/CPAI09/, 2013: Fourth International Constraint Solver Competition.

[60] http://www.emn.fr/z-info/choco-solver/, 2013: Choco Constraint Solver library.

[61] http://www.frantzen.info/index.php?/pages/treeSolver.html, 2013: treeSolver.

[62] http://www.smtlib.org/, 2013: SMT-LIB.

[63] http://z3.codeplex.com/, 2013: Microsoft Z3 SMT-solver.

[64] Wotawa F., Nica M., Aichernig B., 2010: Generating Distinguishing Tests Using the Minion Constraint Solver, in: *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, S. 325–330.