Development and Analysis of Decision Heuristics for an Interval Constraint Solver Handling Non-linear Arithmetic

Diploma Thesis

Linus Feiten

April 2010

Department of Computer Science, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 51, 79110 Freiburg im Breisgau

Foreword

I would like to thank all groups and individuals who supported me during my Computing Science studies leading to the completion of this thesis. First of all my parents, without whose ongoing support and understanding my academic and private path of life would not have been possible in the way it has been. Then, my sister, brother and the whole extended family, whose backing has consciously but probably even more subconsciously carried me through these years.
A dept of gratitude is owed to my friends and fellow students in Freiburg: Andy, Alex, Ben, Clemens, Dosn, Henni, Jakob, Jan, Kosta, Moritz, Najib, Nick, Philip, Sebastian, Seppo and Stefan, who went this way or parts of it with me together. Many thank also go to my friends in Aberdeen, who warmly welcomed me there during my Erasmus year. It is owed to them that I feel fit to write this thesis in English.
I thank Prof. Bernd Becker and Prof. Christoph Scholl for excellent academic teaching and for having invoked my interest for Computer Architecture in the first place, and my supervisors Stefan Kupferschmid and Tobias Schubert, without whose commitment and catching enthusiasm this thesis would not have been possible. Heartfelt thanks are also owed to Dr. Ilia Polian for outstanding teaching and for being of invaluable help together with Dr. Sudhakar Reddy in enabling my internship with Mentor Graphics India. Finally thanks to all staff members of IRA and ABS in Freiburg, who have always been most friendly and cooperative!

Freiburg im Breisgau, April 2010 *Linus Feiten*

Abstract in German

Bei iSAT [1, 9, 10] handelt es sich um einen SMT-Solver, der Boolsche Kombinationen von linearen und nichtlinearen Constraints auf ihre Erfüllbarkeit prüft. Die Variablen in den Constraints können Boolsche, ganzzahlige oder reelwertige Variablen sein und haben zu jedem Zeitpunkt der Berechnungen ein abgeschlossenes Interval zugewiesen. Die Constraints können transzendentale Funktionen enthalten, wodurch es sich um eine im allgemeinen unentscheidbare Theorie handelt. Anders als ein lediglich mit Boolschen Variablen operierender SAT-Solver, kann iSAT daher das Ergebnis "unknown" liefern. Das Verfahren zur Lösung entspricht dem DPLL-Algorithmus [7, 8], aber das Propagieren von Variablenbelegungen ist auf die Intervallarithmetik erweitert. Da die Variablen keine Punktwerte sondern Intervale haben, können sie im Verlauf des Lösungsfortschittes mehrfach entschieden werden.
In dieser Arbeit wurden verschiede Heuristiken entwickelt, implementiert und getestet, welche die Reihenfolge festlegen, in der die Variablen entschieden werden. Diese Reihenfolge hat einen großen Einfluss darauf, wie schnell eine Lösung gefunden wird. Als Resultat steht eine große Auswahl an neuen Heuristiken und Zusatzoptionen zur Verfügung, von denen sich bestimmte als sehr vorteilhaft erwiesen haben, um die benötigte Gesamtlaufzeit zur Lösung der 40 Testbenchmarks um annähernd 50% zu beschleunigen.

Abstract

The solver iSAT [1, 9, 10] is an SMT solver, which computes the satisfiability of Boolean combinations of linear and non-linear constraints. These constraints can contain Boolean, integer or real valued variables, which have bounded intervals assigned to them at any step of the algorithm. Furthermore, the constraints can contain transcendental functions, which makes the theory generally undecidable. Thus, iSAT might also return the result "unknown". The solving procedure is equivalent to the DPLL algorithm [7, 8]. But the propagation of variable assignments is extended to handle interval arithmetic. As variables have no single values but instead intervals assigned to them, they can be decided several times in course of the solving process.
In this thesis, several different heuristics were developed and tested, which determine the order in which the variables are decided. This order can have a great impact on the runtime of the solver. As a result, a big range of new heuristics and additional options is available, some of which have proven to be very beneficial to reduce the overall runtime of 40 tested benchmarks by almost 50%.

Contents

Chapter 1

Introduction

The Boolean satisfiability problem (SAT) is of great importance in contemporary electronic design automation [4, 19] but also in AI planning [14, 15] and other fields where NP-complete problems need to be solved in reasonable time. This is due to the fact that every NP-complete problem can be transformed with only polynomial effort into a Boolean formula whose satisfiability is equivalent to the solution of the respective NP-complete problem. In the nineties of the 20th century, there has been a big leap in the field of so called SAT solver programs, which allow for the handling of bigger and bigger SAT instances (i.e. Boolean formulas). The continuing mission of the SAT community is to develop new SAT solvers which can solve the SAT problem faster and handle bigger SAT instances.

An extension of Boolean SAT is SAT modulo theories [2]. There, instead of only Boolean variables, it is possible to have constraints within the formula which belong to a certain background theory. According to this background theory, each such atom can be evaluated to *true* or *false* under certain circumstances. (If the background theory is undecidable, the constraints can also remain neither *true* nor *false*.) Such atoms can be handled like Boolean variables by a SAT solver algorithm. One way of applying SAT modulo theories with great practical relevance is to take the undecidable domain of non-linear constraints involving transcendental functions as background theory. A constraint can thus be any equation or inequation consisting of variables and real numbers like for example $x < 2 * y^7 + 5$. Whether this constraint evaluates to *true*, *false* or neither depends on the intervals which restrict the values of the variables it contains.

The solver iSAT [1, 9, 10] is a program which tries to compute for a given Boolean combination of non-linear constraints if there exists a value assignment for the occurring variables, such that the whole formula evaluates to *true*. If non such assignment exists, iSAT returns that the formula is unsatisfiable. As the theory of non-linear constraints involving transcendental functions is not decidable it is also possible that iSAT returns "unknown" as a result. The way the algorithm works is that at each step a variable is chosen for a possible new assignment of its interval. The act of assigning a new interval to a variable is called a decision.

The order in which variables are chosen for decisions has a great impact on the time it takes to come to a solution. A heuristic according to which the next decision variable is chosen is called a *decision heuristic*. For pure Boolean SAT, decision heuristics have already been

researched to a degree at which a certain type of heuristic has emerged as yielding the best empirical results. It is, however, not necessarily the case that an adoption of this heuristic type will also work best for a SAT modulo theory solver like iSAT.
In course of the work for this thesis, we invented and implemented several different types of decision heuristics for the interval-based non-linear arithmetic domain, which will be presented in Chapter 3 and 4. As an introduction, Chapter 2 will include the presentation of different topics whose concepts are necessary for understanding the rest of this work. Chapter 5 will present and interpret the empirical results gathered through the final work phase and Chapter 6 will consist of a summary and future prospects.

Chapter 2

Fundamentals

2.1 Propositional logic formulas

To understand the terminology used in this thesis, some foundational aspects of Boolean formulas need to be introduced. Most of the time we will say formula or Boolean formula when we mean propositional logic formula. Furthermore, we will use $\wedge$ and $\vee$ for the propositional connectives *and* and *or* and we will use $\overline{x}$ for the negation of any propositional expression x. An example for such a formula over a set of variables $V = \{a, b, c\}$ could be $f = (a \wedge \overline{b}) \vee \overline{(\overline{a} \vee c)}$.
An assignment for a formula is a function $V \rightarrow \{true, false\}$ that assigns *true* or *false* to each variable $v \in V$. For such an assignment the propositional expression $(x \wedge y)$ is *true*, if and only if both x and y are *true*. If either x or y is *false*, $(x \wedge y)$ is *false*. Similarly, $(x \vee y)$ is *false*, if and only if both x and y are *false*. If either x or y is *true*, $(x \vee y)$ is *true*. The negation $\overline{x}$ of a propositional expression x is *true*, if and only if x is *false*; and *false*, if and only if x is *true*.
To decide whether there exists an assignment such that any Boolean formula evaluates to *true* is known to be NP-complete [6]. This means that there is no known algorithm to produce an answer and to solve the so called SAT (for satisfiability) problem in polynomial time. However, there have been major progresses beginning in the nineties of the 20th century regarding SAT solver programs, which use elaborated search space pruning, learning and branching techniques, such that results for many SAT instances can be computed in reasonable time despite NP-completeness [11, 18, 20, 23].
For most modern SAT solvers the formula has to be in conjunctive normal form (CNF). A CNF is a conjunction of clauses (i.e. clauses connected by $\wedge$'s) and a clause is a disjunction of literals (i.e. literals connected by $\vee$'s). A literal is a variable v in its positive phase v or in its negative phase $\overline{v}$. A literal can be

- *true* $\Leftrightarrow$ The literal is in positive phase and its variable is assigned *true* or the literal is in negative phase and its variable is assigned *false*.
- *false* $\Leftrightarrow$ The literal is in positive phase and its variable is assigned *false* or the literal is in negative phase and its variable is assigned *true*.
- *unassigned* $\Leftrightarrow$ The variable of the literal is unassigned.

A clause can be

- *true* or *satisfied* $\Leftrightarrow$ At least one literal in the clause is *true*.
- *false* or *unsatisfied* $\Leftrightarrow$ All literals in the clause are *false*.
- *unresolved* $\Leftrightarrow$ At least one literal in the clause is *unassigned* and no literal in the clause is *true*.

In order for a CNF to be satisfied, every clause in it has to be satisfied. This means that in each clause there has to be at least one *true* literal.
Every propositional logic formula can be transformed into an equivalent CNF through expansion and application of the boolean distributive law. That, however, has in the worst case an exponential complexity and is therefore not efficiently applicable. But there is a method with only polynomial complexity, which transforms any formula f into a new CNF with the same satisfiability properties as f. And the size of the new formula only grows linearly in the size of the original one [22]. To achieve this, new auxiliary variables have to be added to the variables of the original formula. The formula $f = (a \wedge \overline{b}) \vee \overline{(\overline{a} \vee c)}$, for example, can be transformed into the following CNF using the additional auxiliary variables $U = \{d, e, f\}$. And the process of its creation ensures that this new formula $\textsc{Cnf}(f, U)$ is satisfiable if and only if the original formula f is satisfiable.

$$\begin{aligned}\textsc{Cnf}(f, U) = d \wedge &(d \vee \overline{e}) \wedge (d \vee f) \wedge (\overline{d} \vee e \vee \overline{f}) \\ \wedge &(e \vee \overline{a} \vee b) \wedge (\overline{e} \vee a) \wedge (\overline{e} \vee \overline{b}) \\ \wedge &(f \vee a) \wedge (f \vee \overline{c}) \wedge (\overline{f} \vee \overline{a} \vee c)\end{aligned}$$

In this example, $\textsc{Cnf}(f, U)$ can be reduced even further without changing its satisfiability. This allows us to introduce another concept which is relevant for the work presented in Chapter 4.3. It is that a clause can be *implied* by another clause[1]. Consider two clauses c_1 and c_2 of a CNF and let c_1 include (among others) all literals which are included in c_2:

$$c_1 = c_2 \vee R_1$$

with R_1 being the rest of c_1. If this is the case, we say "c_2 implies c_1" or "c_1 is implied by c_2". Let for example c_1 and c_2 be the following clauses:

$$c_1 = (a \vee \overline{b} \vee c \vee \overline{d} \vee \overline{e}), \qquad c_2 = (a \vee \overline{b} \vee \overline{d})$$

Then R_1 would be $(c \vee \overline{e})$ and we say "c_1 is implied by c_2". It is obvious that every assignment satisfying c_2 automatically satisfies the implied clause c_1 as well. Hence, we do not alter the satisfiability of a CNF, if we remove all implied clauses.
In the example above, both $(d \vee \overline{e})$ and $(d \vee f)$ are implied by the single literal clause d. Thus, the CNF can be simplified even more to:

$$\textsc{Cnf}(f, U) = d \wedge (\overline{d} \vee e \vee \overline{f}) \wedge (e \vee \overline{a} \vee b) \wedge (\overline{e} \vee a) \wedge (\overline{e} \vee \overline{b}) \wedge (f \vee a) \wedge (f \vee \overline{c}) \wedge (\overline{f} \vee \overline{a} \vee c)$$

[1] In other literature the concept of clauses implying other clauses is called *subsumption*. Instead of "c_2 implies c_1" these authors say "c_2 (syntactically) subsumes c_1" or accordingly "c_1 is (syntactically) subsumed by c_2". We have chosen the terminology of implication.

2.2 Boolean SAT and the DPLL algorithm

This section is supposed to give the reader an idea of several terms from the field of Boolean SAT solvers, which will be picked up on several occasions when our heuristics are explained in Chapter 3. These terms will in particular be: decision, decision level, unit clause, Boolean constraint propagation, implication, implication graph, conflict, conflicting clause, learned conflict clause, conflict-driven learning, and backtracking.
It would go beyond the scope of this work to fully cover all possible features and variations of the following techniques. The aim of this section is merely to evoke an intuition, which suffices to understand what follows in the remainder of the thesis. All terms and techniques shall be demonstrated on the basis of one single example. Let the CNF f be given as:

$$f = c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6 \wedge c_7 \wedge c_8$$

With its clauses $c_1, ..., c_8$ being:

$$\begin{aligned}
c_1 &= (x_1 \vee x_2) \\
c_2 &= (\overline{x}_3 \vee \overline{x}_5) \\
c_3 &= (\overline{x}_4 \vee \overline{x}_7) \\
c_4 &= (x_6 \vee \overline{x}_7) \\
c_5 &= (\overline{x}_2 \vee x_6 \vee x_7) \\
c_6 &= (x_3 \vee \overline{x}_4 \vee x_7) \\
c_7 &= (x_4 \vee \overline{x}_6 \vee \overline{x}_7) \\
c_8 &= (\overline{x}_2 \vee x_4 \vee \overline{x}_6 \vee x_7)
\end{aligned}$$

Thus, there are 7 Boolean variables, each of which can be assigned the value *true* or *false*. In order for f to be satisfied, at least one literal in each clause has to evaluate to *true*. The question is: Does an assignment for all variables exist, such that all clauses $c_1, ..., c_8$ are satisfied? A naive way to come to an answer would be to try all possible assignments. As each variable can assume two different values (*true* or *false*), there are $2^7 = 128$ different possible assignments. These can be represented as the leaves of a binary tree, in which each node represents the decision of a single variable. The left branch of a node stands for assigning *false* and the right branch for assigning *true* to the respective variable. Figure 2.1 shows such a tree for only 5 variables. The structure of the tree is referred to as the search space.
The number of possible assignments is always 2 to the power of the number of variables; it grows exponentially with the number of variables. As a consequence, it is always possible to create problems which cannot be solved in humanly time, no matter how fast a physical computer is. That is, if the algorithm is to simply try out all the $2^{\#\text{variables}}$ assignments. Finding a satisfying assignment for a CNF belongs to the problem class of so called NP-complete problems [6]. Expressed in a simplified way this means, that a *non-deterministic*

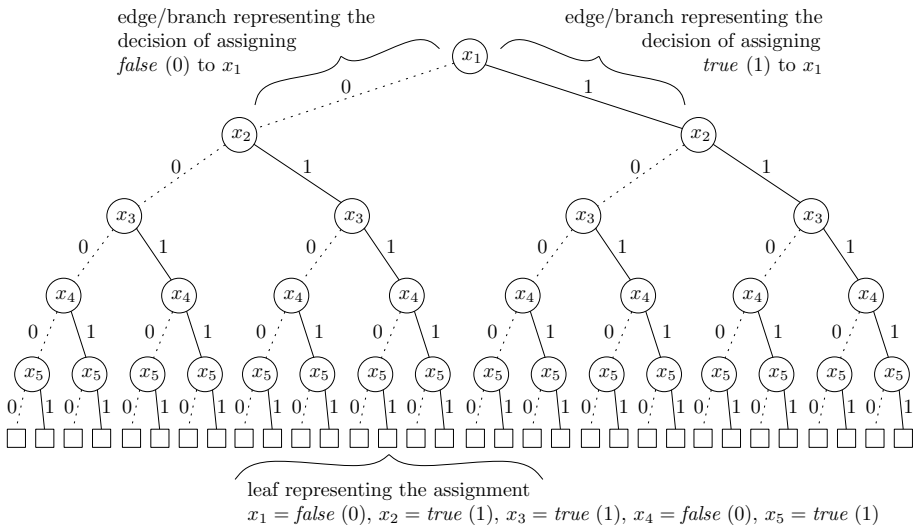

Figure 2.1: A complete binary tree representing the search space in the search for an assignment with 5 Boolean variables. One leaf is picked as a demonstration example of how leaves represent assignments.

algorithm can find a solution in polynomial time. Being non-deterministic the algorithm "guesses" (or knows from an oracle) a suitable path from the root of the binary tree (search space) to one of the leaves which stands for a satisfying assignment. And the check if this leaf is representing a satisfying assignment can be done in polynomial time. But in real physical computers, only deterministic algorithms are possible. Hence, the "parallel" checks of *all* possible assignments done by the theoretical non-deterministic machine are not possible in reality.

Even though it has not yet been proven that a deterministic algorithm for solving NP-complete problems cannot exist, most people believe that there is no way of solving them in polynomial time with a deterministic machine, because such an algorithm has not yet been found. So the only chance of getting a solution for the SAT problem seems to be the expensive way of trying out all possible assignments.

In recent years, however, there have been great achievements in the research of SAT solving algorithms deploying techniques which allow them to prune great portions of the search space, such that many possible assignments do not need to be considered at all [11, 18, 20, 23]. How this is possible shall be shown on the basis of our above example, which we are now coming back to.

The DPLL algorithm is named after its inventors Davis, Putnam, Logemann, and Loveland [7, 8]. Its most basic functionality can be declared with the following simple steps:

1. Decide an unassigned variable, i.e. assign the value *true* or *false* to it.
2. Check if assignments of other unassigned variables have to be made in order for the whole CNF not to become *false*. This is referred to as the deduction step or also Boolean constraint propagation (BCP) or unit-propagation.
3. Check if the following conditions apply and take the appropriate actions:
   - If at least one clause of the CNF evaluates to *false* under the current assignment, undo the variable decision causing this conflict and try the respectively other value for this variable; unless both values have already been tried, in which case an even earlier decision needs to be undone.
   - If all values for all variables have been tried, return "unsatisfiable" (*unsat*).
   - If all variables are assigned and the CNF is not *false*, return "satisfiable" (*sat*) and the current assignment as satisfying assignment.
   - Otherwise, go back to step 1 and decide the next unassigned variable.

Let us assume that the DPLL algorithm working in our example decides the variables in ascending order of their indices: first x_1, then x_2, and so forth. Let us furthermore assume, that at each decision the first value to be tried for an unassigned variable is the value *false*. Hence, the first act of our example solver is to assign *false* to x_1.

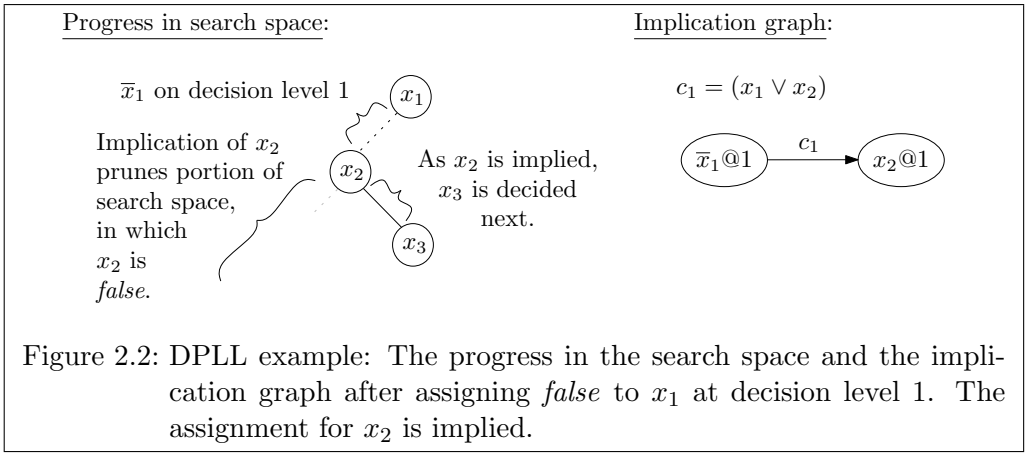

Figure 2.2: DPLL example: The progress in the search space and the implication graph after assigning *false* to x_1 at decision level 1. The assignment for x_2 is implied.

This means we go down the left branch from the root node, which represents the decision of x_1 in our search space tree (see the left side of Figure 2.2). As this has been the first decision, we say that it has taken place at decision level 1. Next, the SAT solver enters step 2, in which it deduces the implications[2] occurring due to the previously made decision, if there are any.

[2]The term "implication" in this context is not to be mistaken with what is meant, when one clause is implied by another clause, as explained on page 4.

In our set of clauses listed on page 5, we see that the clause $c_1 = (x_1 \vee x_2)$ only consists of two literals, one of which is x_1. Now, that x_1 has been assigned *false*, the literal x_1 evaluates to *false* as well. This leaves the clause c_1 with only one unassigned literal, namely x_2. In order for c_1 to be satisfied, the literal x_2 has to evaluate to *true*, because otherwise, with x_1 already being *false*, c_2 would be *false* and with it the whole CNF. A clause, whose literals are under the current assignment all but one *false* and whose remaining literal belongs to an unassigned variable, is called a unit clause.
Every unit clause causes what is called an implication. In our case, c_1 has become a unit clause after the first decision. Hence, the decision of assigning the value *false* to x_1 at decision level 1 implies that the value *true* is assigned to x_2. This is commonly illustrated in an implication graph [18, 24] (see the right side of Figure 2.2).
An implication graph is a directed graph, in which each node represents the assignment of a variable. Each edge from one node to another represents that the assignment of the source node implied the assignment of the sink node. The label of an edge designates, which clause was responsible for the implication by becoming a unit clause after the source node assignment. For readability reasons, we are using the following notation in our implication graph: If the value *true* is assigned to a variable x_i at decision level k, we write $(x_i@k)$. If *false* is assigned, we write $(\overline{x}_i@k)$.
Thus, our implication graph has two nodes after the first decision: One node is $(\overline{x}_1@1)$, which stands for the decision of assigning *false* to x_1 at decision level 1. It does not have any predecessor nodes, as it was a decision and hence not implied by any other assignment. The other node $(x_2@1)$ has $(\overline{x}_1@1)$ as its predecessor. Their connecting edge is labelled with c_1 representing that assigning *false* to x_1 has turned c_1 into a unit clause, which in turn implied the assignment of *true* to x_2 at decision level 1.
As assigning *false* to both x_1 and x_2 will make the whole CNF *false*, the portion of the search space in which these values are assigned does not have to be explored any further. Because it is not necessary to try assignments for the remaining unassigned variables, when the CNF already evaluates to *false* under the currently assigned variables. The implication of assignments through unit clause deduction has exactly this effect of search space pruning.
No other clause but c_1 is directly affected by the assignment of x_1. The implied assignment of x_2, however, also has effects on $c_5 = (\overline{x}_2 \vee x_6 \vee x_7)$ and $c_8 = (\overline{x}_2 \vee x_4 \vee \overline{x}_6 \vee x_7)$, in which x_2 occurs in negative phase. As x_2 has been assigned *true*, these two occurrences now evaluate to *false*. But c_5 and c_8 are still not unit, because they still have more than one unassigned literal each. Thus, no further implications take place, which is the end of step 2 of our simplified DPLL declaration.
As none of the conditions described under step 3 apply, the algorithm comes back to step 1, in which it decides the next unassigned variable. Actually, x_2 would have been the next variable to decide according to the decision order. But as x_2 has already been assigned through implication, the next unassigned variable to decide is x_3.
Hence, x_3 is set to *false* at decision level 2. The only two clauses in which x_3 occurs are $c_2 = (\overline{x}_3 \vee \overline{x}_5)$ and $c_6 = (x_3 \vee \overline{x}_4 \vee x_7)$. As the occurrence of x_3 in c_2 is in its negative phase,

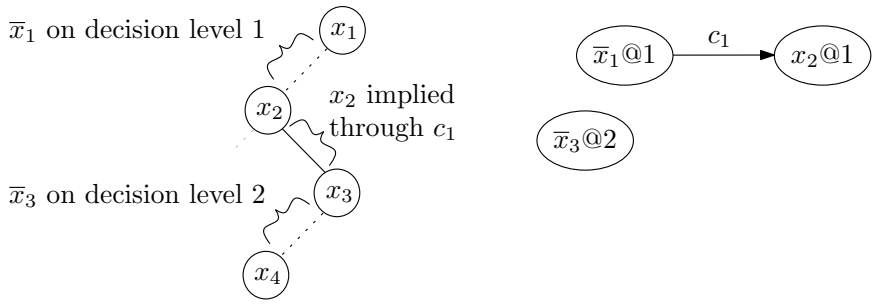

Figure 2.3: DPLL example: The progress in the search space and the implication graph after assigning *false* to x_3 at decision level 2.

this clause is now satisfied. In c_6, x_3 evaluates to *false* but c_6 still has two unassigned literals. Thus, no implication is deduced. Step 3 is also skipped, as none of its conditions are met. We insert the decision of assigning *false* to x_3 at decision level 2 as ($\overline{x}_3$@2) in the implication graph (see Figure 2.3).
The next variable to decide (at decision level 3) is x_4, which is thereby set to *false*. The clauses affected by this decision are $c_3 = (\overline{x}_4 \vee \overline{x}_7)$, $c_6 = (x_3 \vee \overline{x}_4 \vee x_7)$, $c_7 = (x_4 \vee \overline{x}_6 \vee \overline{x}_7)$, and $c_8 = (\overline{x}_2 \vee x_4 \vee \overline{x}_6 \vee x_7)$. As the occurrence of x_4 in c_3 and c_6 is in its negative phase, these two clauses are now satisfied. c_7 and c_8 are left with only two unassigned literals each (x_2 in c_8 is already assigned *true*). So, no unit clauses were generated (condition for step 2), nor has any clause become *false* (condition for step 3). Proceeding to decision level 4, x_5 is assigned *false*. The only occurrence of x_5 is in $c_2 = (\overline{x}_3 \vee \overline{x}_5)$, which was already satisfied due to the assignment of *false* to x_3. The assignments of x_4 and x_5 are illustrated in Figure 2.4.
On decision level 5, x_6 is assigned *false*. This affects $c_4 = (x_6 \vee \overline{x}_7)$, $c_5 = (\overline{x}_2 \vee x_6 \vee x_7)$, $c_7 = (x_4 \vee \overline{x}_6 \vee \overline{x}_7)$, and $c_8 = (\overline{x}_2 \vee x_4 \vee \overline{x}_6 \vee x_7)$. As the occurrence of x_6 in c_7 and in c_8 is in negative phase, these two clauses are now satisfied. c_4 and c_5, however, in which x_6 occurs in positive phase, have both become unit clauses: The only unassigned literals are $\overline{x}_7$ in c_4 and x_7 in c_5. Thus, in order to satisfy c_4, *false* has to be assigned to x_7, and in order to satisfy c_5, it has to be *true*. Each unit clause implies a different assignment! This is what is called a conflict (see Figure 2.5) and c_4 and c_5 are called the conflicting clauses. An analysis of the implication graph shows that the cause of this conflict is not just due to assigning *false* to x_6 but that the conflict might not have occurred, if x_1 was not assigned *false*. Thus, it can be asserted that assigning *false* to x_1 and *false* to x_6 always causes this conflict. To prevent this assignment in the future, a new clause is added to the CNF:

$$c_9 = (x_1 \vee x_6)$$

This clause is called a learned conflict clause. It becomes unit as soon as either x_1 or x_6 is

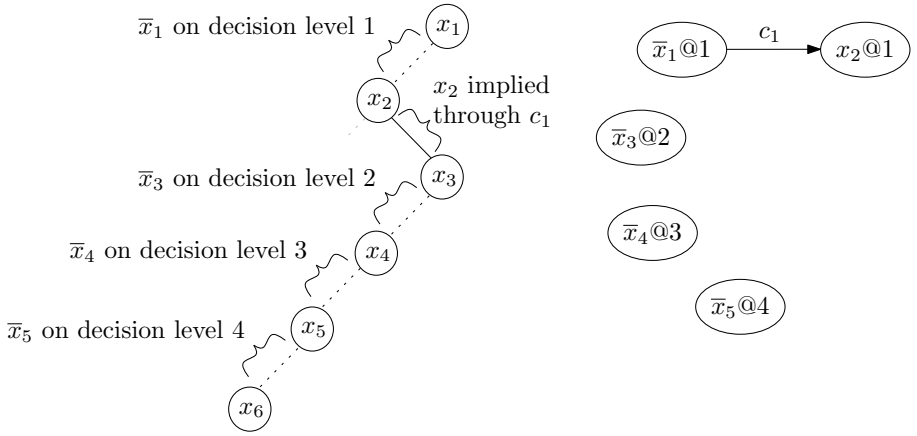

Figure 2.4: DPLL example: The progress in the search space and the implication graph after assigning *false* to x_4 at decision level 3 and *false* to x_5 at decision level 4.

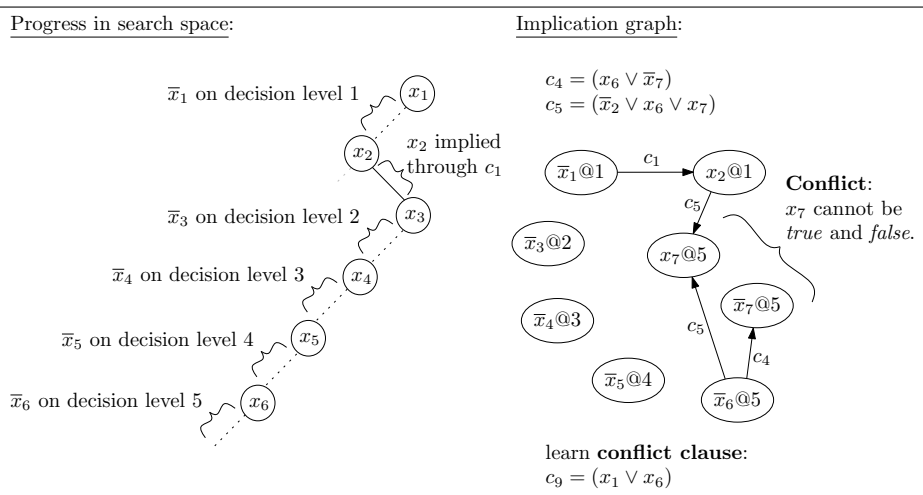

Figure 2.5: DPLL example: Assigning *false* to x_6 at decision level 5 causes a conflict.

assigned *false* and thereby implies that the respectively other variable is assigned *true*.
Conflict-driven learning was first incorporated independently in the SAT solvers GRASP [18] and rel-sat [3]. But the novelty of these solvers was not just the learning of conflict clauses but also a technique called conflict-driven backtracking. Backtracking means that assignments are undone, which were made in previous decisions. The solver jumps back to a previous decision level undoing all assignments made from that decision level on. For that reason, backtracking is also referred to as backjumping in some literature.

2.3 2-watched-literals

The 2-watched-literals method is a technique to quickly identify unit clauses in the Boolean constraint propagation (BCP) phase of the SAT solver. Before this method was invented, there were counter-based solutions, which track the number of literals evaluating to *false* for each clause. If a counter reaches the overall number its clause's literals minus 1, the clause is unit unless the remaining literal evaluates to *true*. Such counter-based BCP methods were employed in solvers such as GRASP [18], rel-sat [3], and satz [16].
The major disadvantage of these techniques becomes apparent when assignments are undone by backtracks. Because then all counters need to be updated, which can become very time consuming. With the 2-watched-literals method, the authors of the solver Chaff [20] introduced a technique to overcome this backtracking update problem. Every clause has two watch-pointers, each of which is initially pointing at a different literal of the clause. The literals pointed at by the watch-pointers are the watched literals of a clause. If a watched literal becomes *false*, the respective watch-pointer has to be set to another literal, which is still not *false*. If none such other literal exists except the other watched literal, the status of the other watched literal has to be checked. If it is *true*, the clause is already satisfied and cannot be unit. If it is unassigned, the clause is unit. And if it evaluates to *false* as well there is a conflict. (The latter case can occur when both watched literals become *false* in course of a single decision.)
When a backtrack occurs and variable assignments are undone, the watch-pointers do not need to be changed, because the watched literals can only change back to being unassigned. If both watched literals were *false*, they caused the conflict and the backtrack will make at least one of them unassigned again. Otherwise, they remain valid watched literals anyway, because one of them is unassigned or *true*.
Let us consider $c_8 = (\overline{x}_2 \vee x_4 \vee \overline{x}_6 \vee x_7)$ from the preceding Section and let its watch-pointers initially point at $\overline{x}_2$ and x_4. After some time, let *true* be assigned to x_2. This means that the literal $\overline{x}_2$ becomes *false*. Hence, the watch-pointer cannot remain with $\overline{x}_2$ and has to find another literal, which is still not evaluating to *false*. This could be $\overline{x}_6$, such that x_4 and $\overline{x}_6$ are now watched.
Next, let *false* be assigned to x_4 such that the literal x_4 cannot remain watched. The last non-watched literal not evaluating to *false* is x_7. Thus, $\overline{x}_6$ and x_7 are now watched. Then, let *true* be assigned to x_6 such that there is no other literal left for $\overline{x}_6$'s watch-pointer to

switch to. When the other watched literal x_7 is still unassigned, c_8 is identified as a unit clause.
If a backtrack undoes any assignments affecting c_8, $\overline{x}_6$ and x_7 become unassigned before any other literal of c_8, because their assignments happened on the most recent decision levels. Hence, the watched-pointers can remain with these literals and do not have to be updated in course of backtracks.

2.4 Decision strategies in Boolean SAT

In this section, the reader is introduced to several decision heuristics as the exist for purely Boolean SAT. A basic understanding of these is necessary for comprehending the descriptions of our newly developed decision heuristics for iSAT. Moreover, several of our these new heuristics are equivalents of or were modelled on already existing heuristics in Boolean SAT.
In Section 2.2, the DPLL algorithm was presented which searches for a variable assignment satisfying the given CNF. Every time, when no deductions through unit propagation are possible, a new variable has to be decided. In the example, we determined the order in which the variables were decided according to their indices. But many other methods are thinkable. Such methods are called decision heuristics[3].

2.4.1 Böhm

In early SAT solvers, most decision heuristics counted the occurrences of variables in clauses with certain attributes. Böhm's heuristic [5] counts in how many unresolved clauses of a certain length a variable occurs. Variables occurring in short clauses are preferred, as short clauses are more likely to become unit clauses. The next decision variable is the one currently occurring in most unresolved clauses of the shortest length. And the length of a clause is the number of its unassigned variables. In [17] this is illustrated as a vector $H(x) = (H_1(x), H_2(x), ..., H_n(x))$ for every variable x, where $H_i(x)$ represents the number of occurrences of x in unresolved clauses of length i. The variable with the maximal vector in lexicographic order is selected for decision.
Hence, a variable x occurring in 0 clauses of length 1, in 4 of length 2 and in 0 clauses with a length greater than 2 would have the vector

$$H(x) = (0, 4, 0, 0, ..., 0)$$

In the same way, a variable y occurring in 0 clauses of length 1, in 3 of length 2, in 27 of length 3 and in 0 of a length greater than 3 would have the vector:

$$H(y) = (0, 3, 27, 0, ..., 0)$$

[3] As a decision is always equivalent to choosing a branch in the search space binary tree, the term "branching heuristic" is also used in some literature denominating the same as "decision heuristic".

In this scenario, Böhm's heuristic selects x for decision because x occurs in more clauses of length 2 than y. The fact that y also occurs in 27 clauses of length 3 is not taken into account, which can be considered a flaw of the heuristic. An approach to avoid such potentially disadvantageous decisions is the Jeroslow-Wang heuristic.

2.4.2 Jeroslow-Wang

The Jeroslow-Wang decision heuristics were proposed in [13] and further analysed in [12]. Like Böhm's heuristic it works with counters indicating in how many clauses of a certain length a variable occurs. There are two versions of Jeroslow-Wang: one differentiating between literals in positive and negative phase (one-sided Jeroslow-Wang) and one adding both the occurrences together (two-sided Jeroslow-Wang). In the following we assume that the latter version is used, which is also the one we based our iSAT equivalent on. For each variable x, the following function is calculated:

$$JW(x) = \sum_{i=1}^{m} H_i(x) * 2^{-i}$$

with m being the maximum clause length. $H_i(x)$ is the number of occurrences of x in unresolved clauses of length i as described in Section 2.4.1. Thus, the shorter a clause is in which x occurs, the more will this occurrence add to the sum of $JW(x)$. An occurrence in a clause of length l weighs twice as much as an occurrence in a clause of length $l + 1$. The variable with the greatest JW value is preferred for decision. In this the potential shortcoming of Böhm's heuristic can be avoided.
Revisiting the example from Section 2.4.1 demonstrates this. Let x and y be again the variables, whose occurrences in conflict clauses of certain lengths are represented by the the vectors:

$$H(x) = (0, 4, 0, 0, ..., 0)$$

$$H(y) = (0, 3, 27, 0, ..., 0)$$

Böhm's heuristic prefers x for decision because x occurs in more clauses of length 2 than y. The Jeroslow-Wang heuristic, however, calculates $JW(x)$ and $JW(y)$ as follows:

$$JW(x) = 4 * 2^{-2} = 1$$

$$JW(y) = 3 * 2^{-2} + 27 * 2^{-3} = 4.125$$

Hence, y is be preferred.

2.4.3 DLIS and DLCS

Another way of counting the occurrences of variables in clauses are the **DLCS** and **DLIS** heuristics, which were proposed in [17]. DLCS stands for "dynamic largest combined sum (of literals)" and DLIS for "dynamic largest individual sum (of literals)". Both heuristics

count the occurrences of a variable in unresolved clauses as literals in positive and negative phase.
$C_P(x)$ is defined as the number of occurrences in unresolved clauses of a variable x in positive phase. $C_N(x)$ is this number for occurrences in negative phase. If a variable x has great values for $C_P(x)$ and/or $C_N(x)$, it can be considered a good variable to be decided next. Because many clauses will be affected by this assignment. If x is assigned *true*, all clauses which contributed to $C_P(x)$ will be satisfied, and those which contributed to $C_N(x)$ will come one literal closer to becoming unit clauses. If x is assigned *false*, it is the other way around.
To satisfy clauses or to generate unit clauses can both be considered beneficial for the overall SAT solving process. Hence, variables with great $C_P(x)$ and/or $C_N(x)$ are favoured by DLCS and DLIS. The difference between the two heuristics lies in the "and/or" between $C_P(x)$ and/or $C_N(x)$. DLCS adds (hence combined sums) both values, whereas DLIS considers the values individually, such that the variable with the greatest value of either $C_P(x)$ or $C_N(x)$ is preferred.
DLCS and DLIS do not only have an influence on which variables are decided. They also determine which value is assigned to the variable. The authors of [17] were aiming at satisfying as many clauses as possible. Hence, if $C_P(x) \geq C_N(x)$ they assign *true* to x and in case of $C_P(x) < C_N(x)$ they assign *false*.

2.4.4 VSIDS

The heuristics described in the previous sections have a common disadvantage: Similar to the BCP techniques predating 2-watched-literals they need a lot of effort to maintain their counters. Especially when a backtrack takes place, it can be expected, that a lot of counters have to be updated. Thus, the counter maintenance dominates the overall computation time of a solver.
In 2001, the authors of [20] proposed a heuristic called VSIDS, which was able to overcome this problem by making updates in case of backtracks redundant. The acronym stands for "Variable State Independent Decaying Sum". And "state independent" expresses that the state (i.e. the assignment) of a variable has no impact on the variable's position in the decision order.
What VSIDS does, is that at the beginning of the solving process, every variable is initialised with a value corresponding to its number of occurrences in the initial CNF. This is not different from what DLCS does. But the way these values are altered in course of the solving process is very different: VSIDS increases the value of a variable every time it occurs in a newly learned conflict clause. Thus, the variables occurring in most conflict clauses have the greatest values.
Furthermore, the values of all variables are periodically divided by a constant number, such that variables which occurred in recent conflict clauses are weighed more. This is what "decaying sum" in VSIDS stands for. The VSIDS heuristics were shown to generally outperform all previous decision heuristics, which were based on literal counting.

2.5 Bounded Model Checking and SAT

This chapter caters both for giving an example of the practical use of SAT in the industry, and for conveying the concept of Bounded Model Checking (BMC) and SAT. In BMC one is interested in whether a (mostly finite) state system can run into a state in which an invariant is violated. For example, the controller of a train's door must never reach a state in which the train is moving and the door is open. To model this, all states of the system are encoded as vectors of Boolean values. A state a could for example be encoded as the bit vector

$$\overrightarrow{S}_a = (0, 1, 1, 0, 1, \ldots, 1)$$

and a state b as the bit vector

$$\overrightarrow{S}_b = (1, 1, 0, 1, 0, \ldots, 0).$$

After the states are encoded, one declares certain constraints in form of predicates over so called state variables $\overrightarrow{W} = (w_0, \ldots, w_n)$. These state variables act as placeholders, in which the bit values like $(0, 1, 1, 0, 1, \ldots, 1)$ for state a or $(1, 1, 0, 1, 0, \ldots, 0)$ for state b can be inserted. Furthermore, one declares an initial constraint $I(\overrightarrow{W})$, which evaluates to *true* if and only if the state values substituted for $\overrightarrow{W}$ encode an initial state. In the following we will sometime just say: "a predicate 'accepts' a state", to express this.
Besides the initial constraint $I(\overrightarrow{W})$ there is also the final constraint $F(\overrightarrow{W})$, which evaluates to *true* if and only if the state values substituted for $\overrightarrow{W}$ encode a state, in which the desired invariant is violated. In our train example, F would accept states, in which the train is moving and the door is open.
As a third and last constraint, there is the transition constraint $T(\overrightarrow{W}, \overrightarrow{W}')$, which evaluates to true if and only if the system has a transition from the state values substituted for $\overrightarrow{W}$ to the state values substituted for $\overrightarrow{W}'$. The whole model is written as $M = (I, T, F)$. A run of the model M of length $j \leq i \leq k$ is a sequence of states $\overrightarrow{S}_0 \ldots \overrightarrow{S}_j \ldots \overrightarrow{S}_k$ with the following conditions:

- $I(\overrightarrow{S}_0) = true$
  The state $\overrightarrow{S}_0$ is accepted by I.
  I.e. $\overrightarrow{S}_0$ is a valid initial state.

- $\forall 0 \leq i < k : T(\overrightarrow{S}_i, \overrightarrow{S}_{i+1}) = true$
  All states, which stand next to each other in the sequence, are accepted by T.
  I.e. there exists a transition from each state to its successor state in the sequence.

- $\exists j \leq i \leq k : F(\overrightarrow{S}_i) = true$
  There exists at least one state among $\overrightarrow{S}_j, \ldots, \overrightarrow{S}_k$ which is accepted by F.
  I.e. one of these states is a bad state, in which the invariant is violated.

These three conditions can be combined and translated into a single Boolean formula:

$$\textsc{Bmc}_j^k(M) = I(\overrightarrow{W}_0) \wedge \left(\bigwedge_{0 \leq i < k} T(\overrightarrow{W}_i, \overrightarrow{W}_{i+1}) \right) \wedge \left(\bigvee_{j \leq i \leq k} F(\overrightarrow{W}_i) \right)$$

We now have *time step instantiated* versions $\overrightarrow{W}_i$ of our general variables $\overrightarrow{W}$. This means that the instantiated variables $\overrightarrow{W}_i$ act as place holders for states of M at the specific time step[4] i.
If $\textsc{Bmc}_j^k(M)$ is satisfiable, there exists a run of length i with $j \leq i \leq k$. This means that it is possible for the system to run into a bad state. And this bad state is one of the states between step j and step k. Furthermore, the satisfying variable assignment reveals, which states were traversed from the initial state to the bad state. This kind of information can be very helpful for debugging the system modelled by M.

2.6 iSAT

In Section 2.2, the Boolean SAT problem was presented and it was shown how it is solved by the DPLL algorithm. This concept can be extended, by replacing the Boolean literals in the CNF with another form of expressions from a certain background theory. This is referred to as the paradigm of Satisfiability Modulo Theories (SMT, e.g. [2]). For the iSAT algorithm [1, 9, 10], the background theory is the domain of non-linear constraints involving transcendental functions. This domain is generally undecidable, such that iSAT does not always return *satisfiable* or *unsatisfiable* but can also have *unknown* as a result.

2.6.1 Variable intervals and constraints

A definition of constraints in iSAT can be given as follows:

constraint	::=	*pair* | *triplet*
pair	::=	*variable relation unary_operator operand*
triplet	::=	*variable relation operand binary_operator operand*
relation	::=	$< \mid \leq \mid = \mid \geq \mid >$
unary_operator	::=	exp | log | abs | sin | cos | power | sqrt
binary_operator	::=	+ | - | * | min | max
operand	::=	*constant* | *variable*

The term *constant* can be replaced by any rational number. The term *variable* is a placeholder for variables or powers of variables. In iSAT, variables are not restricted to being Boolean but can also be integer or real valued. All variables have to be assigned a bounded interval at any time, particularly at the beginning of the algorithm.

[4] In other literature a "step" of a model or system is also referred to as a "cycle". We will, however, stick to the "step" terminology.

A constraint can be

- *true* or *satisfied* $\Leftrightarrow$ The constraint (in-)equation is *true* for all values within its variables' intervals.
- *false* or *unsatisfied* $\Leftrightarrow$ The constraint (in-)equation is *false* for all values within its variables' intervals.
- *consistent* $\Leftrightarrow$ Within the variables' intervals there are both values for which the constraint (in-)equation is *true* and values for which it is *false*.

The constraint $(x = 4)$ with $x \in \mathbb{R}$, for example, only evaluates to *true*, if and only if the point interval $[4, 4]$ is assigned to x. If the upper bound of the interval is less than 4, for example $x \in [2, 3]$, the constraint $(x = 4)$ evaluates to *false*, because no value in $[2, 3]$ can satisfy $(x = 4)$. Analogously, the constraint evaluates to *false*, if the interval's lower bound is greater than 4, for example $x \in [5, 6]$. If the upper bound is greater than 4 and the lower bound less than 4, for example $x \in [3, 5]$, the constraint $(x = 4)$ is *consistent*. Because in the interval $[3, 5]$ there are still – but not exclusively – values which can satisfy $(x = 4)$.
The bounds of an interval can be strict or non-strict. In the above example, the bounds were both non-strict, which means that the values of the upper and lower bounds themselves are part of the interval. Non-strict bounds are expressed through the "[ ]"-bracket notation. Thus, $(x = 4)$ is *consistent* under the assignment $x \in [3, 4]$, because the satisfying value 4 is included in the interval. If a bound is strict, its value is not part of the interval. Strict bounds are expressed through the "( )"-bracket notation. Thus, $(x = 4)$ evaluates to *false* for $x \in [3, 4)$, because 4 is not included in the interval.
When iSAT is given the input data of a problem, the constraints are generally not written in the format defined above. Their structure can be much more complex like for example:

$$(x^3 + sin(2 * y) \leq 3 * ln(z) + 0.5 * z)$$

with $x, y, z \in \mathbb{R}$. Such constraints cannot be processed by iSAT directly. They have to be rewritten into the pair/triplet format before the actual solving process starts. This is done using additional auxiliary variables similar to the ones used for Tseitin transformation. Thus, this example constraint is transformed into the conjunction of the following pair/triplet constraints using the auxiliary variables $h_1, ..., h_7$:

$$\begin{array}{ll}
(h_1 \leq 0 + h_2) & (triplet) \\
(h_1 = x^3 + h_3) & (triplet) \\
(h_3 = sin(h_4)) & (pair) \\
(h_4 = 2 * y) & (triplet) \\
(h_2 = h_5 + h_6) & (triplet) \\
(h_5 = 3 * h_7) & (triplet) \\
(h_7 = ln(z)) & (pair) \\
(h_6 = 0.5 * z) & (triplet)
\end{array}$$

2.6.2 The algorithm

The solving process of iSAT is very similar to Boolean DPLL an can be drafted in the following pseudo code:

```
Data : CNF F
Result : sat, unsat or unknown
 1:   while true do
 2:       if propagate() then
 3:           if !decideVar() then
 4:               return sat/unknown;
 5:           end
 6:       else
 7:           if !analyseBacktrack() then
 8:               return unsat;
 9:           end
10:       end
11:   end
```

First, propagate() is executed (line 2). If it succeeds, decideVar() is executed (line 3). If decideVar() fails, a solution can be returned (line 4). If it succeeds, propagate() is executed again (line 2). If propagate() fails (line 6), there is a conflict and analyseBacktrack() is executed (line 7). If analyseBacktrack() fails, the given formula F is unsatisfiable (line 8). If it succeeds, the assignments causing the conflict have been undone and propagate() is executed again (line 2).

propagate()

The iSAT equivalent of Boolean constraint propagation (BCP) is called interval constraint propagation (ICP), which takes place in the propagate() function. In BCP, unit clauses are identified and variable assignments are implied to satisfy them. ICP works similarly, as it also identifies unit clauses and implies variable assignments, such that the remaining *consistent* constraint of a unit clause becomes *true*.
Let $((x < -1) \vee (x^2 = y))$ be a clause in a CNF handled by iSAT and let the following variable intervals be assigned: $x \in [2, 13]$ and $y \in [0, 100]$. This makes the constraint $(x < -1)$ *false* while $(x^2 = y)$ is still *consistent*. Thus, the clause is unit and variable assignments have to be implied which satisfy this *consistent* constraint. Through ICP it is found that the upper bound of x must not be greater than 10, because $10^2 = 100$ which is the upper bound of y. Analogously, the lower bound of y must not be less than 4, because $\sqrt{4} = 2$ which is the lower bound of x. So, new interval assignments are implied: $x \in [2, 10]$ and $y \in [4, 100]$.

decideVar()

When the solver has performed all possible implications and the CNF is still not satisfied, a variable has to be decided just as it is the case in Boolean SAT. In the domain of interval constraints, deciding a variable means to change (i.e. reduce) its interval. A default way of doing this is to split the interval in half, but there are other possibilities as well (see Section 4.2). Thus, an integer or real valued variable can be decided several times; unlike Boolean variables which can only be decided once.

returning unknown

In line 4 of the iSAT pseudo code it says "return sat/unknown". In the following we explain what the result "unknown" means. As a matter of fact, iSAT cannot prove in many cases that a given formula is satisfiable. Because many constraints, cannot become *true*, as the values needed to achieve this are not representable.
Take for example the constraint $(h_3 = sin(h_4))$ and let the current interval of h_3 be the point interval $[1, 1]$. In order for $(h_3 = sin(h_4))$ to be satisfied, h_4 has to assume the point interval of $\pi/2$ or $(4n + 1) * \pi/2$ with $n \in \mathbb{N}_0$. But as iSAT does not support symbolic representation of numbers, π cannot be represented. What iSAT does instead, is to split the interval of h_4 again and again until its upper and lower bounds have come as close to π as the minimal splitting width (MSW) allows. The MSW parameter of iSAT sets the value, down to which the width of an interval can be split. This is necessary for termination. When MSW is reached for all variables and the CNF is still not satisfied, iSAT returns "unknown" and gives the intervals as candidate solutions. It is called candidate solution because these intervals may or may not include a satisfying assignment.
But even when there are no transcendental functions involved, the problem can occur. Take for example the fact that the number 0.1 cannot be precisely represented in the IEEE Standard for Floating-Point Arithmetic (IEEE 754). The reason for that is that in the decimal numeral system, which composes numbers as sums of powers of ten, 0.1 stands for $0 * 10^0 + 1 * 10^{-1}$ and can therefore be represented without any period. In the binary numeral system underlying IEEE 754, however, numbers are composed of different powers of two and the following can be observed:

$$\begin{aligned} & (0.0001100)_{binary} = \left(2^{-4} + 2^{-5}\right)_{decimal} = (0.09375)_{decimal} \\ < \ & (0.1)_{decimal} \\ < \ & (0.0001101)_{binary} = \left(2^{-4} + 2^{-5} + 2^{-7}\right)_{decimal} = (0.1015625)_{decimal} \end{aligned}$$

This example demonstrates (albeit does not proof) that the decimal number 0.1 cannot be represented as a binary number without a period. Hence, it cannot be represented in IEEE 754. And this is why a constraint like $(x = 0.1)$ would lead into a situation of the same kind as $(h_3 = sin(h_4))$, because it would never be considered satisfied.

It could be considered very dissatisfactory, that iSAT cannot really show the satisfiability of many problems. But from a practical point of view this might not be that big a downside as it first seems. Remember what has been explained in Section 2.5 about SAT and BMC. Most practical problems, which a solver like iSAT would be applied to, are BMC related. In these scenarios the designer of a system, whose model includes non-linear interval arithmetics, needs to know if the system violates an invariant within the given amount of transition steps. If the solver returns "unsat", it is proven that this is not the case. And "unsat" is a result, which iSAT can reliably provide.
In case the solver returns "sat", the designer knows how the system reached the unsafe state. But when iSAT returns "unknown" with a candidate solution instead of "sat", it can still be quite helpful. One has to take into consideration that the systems examined through BMC most commonly include analogue or non-discrete components like sensors and actuators. These components all have certain blurry or noisy characteristics due to their physical properties. Thus, if iSAT returns a candidate solution, which consists of very small (though not point) intervals, one might already be well advised to change the system's design, such that it cannot even come that close to a solution. Because when there are blurry and noisy components in the real system, these might sometimes just "tip the scales".

analyseBacktrack()

Like in Boolean DPLL, there also exists an implication graph in iSAT, such that it is possible to execute conflict-driven backtracking. This includes the learning of conflict clauses, which are constructed in accordance with the decisions responsible for the conflict. In iSAT, each decision of a variable is a retrenchment of its interval. This means that either a new greater lower bound or a new smaller upper bound is assigned. Let, for example, the interval of a variable x be $[0, 10]$ before it is decided. Then let iSAT make the decision of splitting the interval in half by assigning 5 as the new lower bound. Thus, only $x \in [5, 10]$ is considered for the following branches of the decision tree. This decision can be expressed as a constraint itself, namely $(x \geq 5)$. Now, let the decision of another variable y be the assignment of a new *strict* upper bound 4. This can be expressed as the constraint $(y < 4)$. And finally let these two decisions be responsible for a conflict. So, the analysis of the implication graph yields that the decisions $(x \geq 5)$ and $(y < 4)$ always lead to a conflict. To avoid this combination of assignments in the future, the conflict clause $((x < 5) \vee (y \geq 4))$ is learned and added to the CNF. Now, $(x \geq 5)$ implies $(y \geq 4)$, and vice versa $(y < 4)$ implies $(x < 5)$.
This small example makes clear how conflict clauses in iSAT are created and why they only consist of constraints of the following form: $(v \circ c)$, with v being a variable, c being a constant value and $\circ$ being a relation $<, \leq, \geq$ or $>$. These kinds of constraints are called *simple bounds*. The fact that conflict clauses only consist of simple bounds is of great relevance for our considerations described in Section 3.3.

Chapter 3

iSAT Decision Heuristics

This chapter consists of detailed descriptions for all heuristics we implemented for iSAT. Each heuristic and, if applicable, its parameters are explained as well as the respective intuitions behind them. As a matter of course, it is not possible to derive an optimal heuristic by purely mathematical means. Contrariwise, for each heuristic it is always possible to construct a problem such that the variables preferred by this heuristic will result in a maximum execution time. Hence, our approach had to be a rather engineered one than purely mathematical. Our method was to think of heuristics whose functionalities can be conceived as potentially beneficial for the SAT solving process.

3.1 Heuristics deciding by variable attributes

This section is about the category of iSAT decision heuristics, which prefer a variable according to certain static attributes. Static means that these variable attributes never change during the SAT solving process.

3.1.1 BMC-forward/backward (bmc-fwd, bmc-bwd)

In Section 2.5, the basic concepts of bounded model checking (BMC) were explained and how they are related to SAT solvers. It has been shown that the formula representing the BMC problem consists of variables whose assignments encode the states of the examined system. Depending on the depth of the formula each of these variables occurs several times in the formula with different indices. Each index stands for a step of the system.
Hence, there are state variables with the index 0 representing the initial states of the system. Then, there are variables with the index 1 representing the states following the initial state, and so forth. This goes on until the maximum depth of the formula is reached. If the depth is n, the index of the state variables cannot be greater than n. The variables with index n represent the last considered step of the system.
The aim of BMC is to identify, whether an invariant is violated at a step $k \leq n$ of the system. The BMC formula is constructed such that the SAT solver will find a satisfying variable assignment if and only if the system can run into a bad state in k steps. This assignment is called a counter example. If the SAT solver returns “unsat”, it is proven that the invariant cannot be violated at any step of the system until step n. In order to

prove which states can or cannot be reached by the system in k steps, it is necessary to make assignments for all variables with indices $\leq k$.
The BMC-forward heuristic prefers variables depending on how small their BMC index is. Thus, variables with a small index are decided first. This makes sense because in order to know whether a state at step k is reachable, the states at the steps $< k$ have to be known as well. Furthermore, it is probable that assigning variables of earlier steps leads to ICP implications, which narrow down the decision possibilities for later steps. This would decrease the search space for the rest of the solving process. If a counter example exists for a rather early step, the assignments for variables with greater indices are not even necessary.
In [21] Strichman puts forward that it is quite possible to assign variables belonging to different ranges of indices without the SAT solver realising that these assignments are already contradicting each other. It could, for example, be the case that all variables with indices between 4 and 12 are already assigned and so are all variables with indices between 14 and 18. To the SAT solver, the assignments are still consistent until it tries to assign the variables with the index 13, which reveal that there is no transition between the currently assigned index 12 and index 14 states. So the many of the previous assignments have been tried in vain.
Thus, it can be considered beneficial to assign the variables in the order of their indices such that "gaps" between assigned variables cannot occur. In this context it does not matter whether the variables are assigned beginning with the greatest or the smallest indices. The "gap" avoiding effect will be the same in both alternatives. Hence, we do not just have a BMC-forward heuristic, which decides variables with small indices first, but also a BMC-backward heuristic, which prefers variables with great indices instead.

3.1.2 Dominant-first/last (dominant-fst/lst)

In Section 2.6 it has been shown how new auxiliary variables are introduced, when iSAT is rewriting complex constraints from an input problem definition into its pair/triplet format. Let for example the following constraint be part of a problem definition given to iSAT:

$$(3 + x) * 0.5 - 7 * y \leq 5$$

As this is not in accordance with pair/triplet format, the following triplets will be created using h_1, h_2 and h_3 as new auxiliary variables:

$$h_1 = 3 + x \qquad h_2 = h_1 * 0.5 \qquad h_3 = 7 * y \qquad h_2 - h_3 \leq 5$$

We call variables which are not auxiliary dominant. In this example, x and y are the dominant variables. The dominant-first heuristic gives preference to dominant variables. Because its intuition is that the problem space is in fact spanned by the dominant variables, whereas the auxiliary variables are only a result of rewriting the problem. So, in order to explore the whole problem space, it might be beneficial to decide the dominant variables

first. Because the split of a dominant variable is more likely to actually correspond to a spilt of the problem space.

Furthermore, if dominant and auxiliary variables are treated equally by a heuristic, the following side effect can occur: Consider again the example above with the four triplets. The order in which iSAT initially stores the variables is according to when they are created. In the pair/triplet rewriting process, iSAT follows a depth-first/left-first strategy in examining the constraint to be rewritten. Hence, the order of the two dominant variables x, y and three auxiliary variables h_1, h_2, h_3 in this example would be x, h_1, h_2, y, h_3. If these five variables are decided in this order, it would effectively be like deciding x three times in a row and then deciding y two times.

This is because a decision of h_1 or h_2, will through ICP automatically reduce the interval of x. And this would also have been the case if x itself had been decided. Let for example the interval of x be $[-100, 100]$. Because of $(h_1 = 3 + x)$, the interval of h_1 would be set to $[-97, 103]$ by ICP, even before any decision is made. Now, we decide h_1 by splitting its interval in the middle and consider the right portion in the following. I.e. the interval of h_1 becomes $[3, 103]$. The ICP step following this decision, will realize that $(h_1 = 3 + x)$ can only be satisfied if x is in the interval $[0, 100]$. Hence, the interval of x is also reduced like it would have been the case if x had been decided itself. Similarly, a decision of h_2 has through $(h_2 = h_1 * 0.5)$ an ICP effect on the interval of h_1, which in turn will imply a new interval for x.

In the same way a decision of h_3 will reduce the interval of y. So, by deciding the variables x, h_1, h_2, y, h_3 in the order they were created, it would be similar to deciding x three times and then deciding y two times. But why should x be decided so often before y is decided for the first time? Should the two dominant variables x and y not be decided in turns when they are spanning the problem space? The dominant first heuristic would decide the variables in the order x, y, h_1, h_2, h_3 when no other options are activated.

For completeness reasons we also implemented the converse heuristic dominant-last, which prefers auxiliary variables.

3.1.2.1 All-and-only-dominant-first

At a later stage of our research we realised that the implementation of dominant-first might not be exactly according to our initial intuition. Because dominant-first does decide the auxiliary variables as well, after all dominant variables have been decided only once. Only after that the dominant variables are decided again.

The “all-and-only-dominant-first” heuristic is similar to dominant-first. But it does not decide any auxiliary variable as long as there are still dominant variables which have not reached an interval of the minimal splitting width.

3.1.3 Boolean-first/last (b-fst, b-lst)

The CNF to be solved by iSAT can include variables of different types which can be Boolean, integer or real. When the Boolean-first heuristic is used, variables of the type Boolean will be decided first. As Booleans can only have the value 0 or 1 they can only be decided once. So it can be assumed that deciding these first will cut off great portions of the search space at an early stage of the algorithm, before the really time consuming splitting of the remaining variables begins.
Let for example a and b be Boolean variables and let x be a real valued variable with the initial interval $[-1000000, 1000000]$. Furthermore, let the problem contain the following clauses[1]:

$$(b \vee a) \wedge (\overline{b} \vee a) \wedge (a \rightarrow ((x > 1) \wedge (x < 0.1)))$$

It is obvious that these clauses are unsatisfiable, because $(b \vee a) \wedge (\overline{b} \vee a)$ can only be satisfied when *true* is assigned to a. But if a is *true*, $((x > 1) \wedge (x < 0.1))$ must be *true* as well. Otherwise $(a \rightarrow ((x > 1) \wedge (x < 0.1)))$ is not satisfied. $((x > 1) \wedge (x < 0.1))$, however, is unsatisfiable. Hence, the whole CNF containing these clauses is unsatisfiable.
This can be realised by iSAT very quickly, if it decides the Boolean variables a and b first. The reason for this is, that after only one decision iSAT will have deduced though ICP that a has to be *true*, if the decision itself has not been setting a to *true*. This is shown in the following case differentiation:

Case 1: Decide $a = true$:	$a = true$ as decision (trivial).
Case 2: Decide $a = false$:	$(b \vee a)$ implies $b = true$. $(\overline{b} \vee a)$ implies $b = false$. $\Rightarrow$ Conflict leads to backtrack and $a = true$.
Case 3: Decide $b = false$:	$(b \vee a)$ implies $a = true$.
Case 4: Decide $b = true$:	$(\overline{b} \vee a)$ implies $a = true$.

Now, that *true* is assigned to a, the clause $(a \rightarrow ((x > 1) \wedge (x < 0.1)))$ will immediately imply both $(x > 1)$ and $(x < 0.1)$, which is a conflict. As no other assignments for a are possible, iSAT can return "unsat".
If, on the other hand, iSAT follows a heuristic by which it decides real valued variables first, the following occurs: $x \in [-1000000, 1000000]$ is split in the middle. If $[-1000000, 0]$ is considered first, $(x > 1)$ becomes *false* and that $a = false$ is implied because of $(a \rightarrow ((x > 1) \wedge (x < 0.1)))$. This will lead to the same conflict as in *case 2* of the above case differentiation. Thus, after a backtrack is done and $[0, 1000000]$ is considered instead. Now, both $(x > 1)$ and $(x < 0.1)$ are still consistent. So no implications take place and x i

[1] For legibility reasons we will in this example ignore the fact that iSAT would rewrite these clauses into several pair/triplet constraints before actually starting the solving process. Our example works the same if transformed into pair/triplet constraints, but would be very less legible.

decided again. This procedure can repeat itself until the interval will have narrowed down to a size at which one of its bounds lies between 0.1 and 1, because only then iSAT will realize that satisfying $(x > 1)$ and $(x < 0.1)$ at the same time is not possible.

This example may seem somewhat constructed, but the occurrence of such scenarios - on a much more complicated scale of course - cannot be ruled out. It shows clearly how one Boolean decision can avoid a lot of futile non-Boolean decisions.

We also implemented the converse heuristic Boolean-last, which puts the variables in the opposite decision order of Boolean-first. Hence, all non-Boolean variables will be decided first. But after each non-Boolean variables has been decided once, Boolean-last decides the Boolean variables as well. And these can only be decided once anyway.

3.1.4 Integer/Real-first/last (i-fst, i-lst)

Like with the Boolean-first/last heuristics we also wanted to be able to prefer other variable types (integer and real valued) for decision. It cannot be justified a priori, why one of these heuristics should be beneficial. But for completeness reasons we implemented integer-first/last heuristics as well. A real-first/last heuristic did not have to be implemented explicitly, as its functionality can be achieved through combinations of boolean-last and integer-last.

3.2 Heuristics deciding by variable interval

In iSAT every variable has a bounded interval assigned to it at any time. These intervals are initially given through the definition of the formula, which iSAT is solving. During the solving process variables are decided, i.e. their intervals are split at certain points and either the right or left side is considered for the following part of the algorithm. If assignments are undone through backtracks the intervals can become greater again, but in general the variable intervals are getting smaller and smaller the longer the algorithm continues. The intuition behind the heuristics described in this section is that the current size of a variable's interval might correlate with how beneficial its decision is for the solving process.

3.2.1 Small-interval-first/last (si-fst, si-lst)

The small-interval-first heuristic decides variables first, which have the smallest intervals. Small-interval-last works exactly the other way around. Both heuristics take into account the absolute width of a variable's interval. This means that a real valued variable with, for example, an interval width of 1.0 will be treated like an unassigned Boolean variable, whose interval $[0, 1]$ $(false = 0, true = 1)$ also has a width of 1.0.

3.2.2 Relative-small-interval-first/last (si-rel-fst, si-rel-lst)

The relative-small-interval-first/last heuristics were implemented to overcome a potential deficiency of small-interval-first/last. This deficiency is that it does not take into account that a real valued variable can be split much more often than an integer or Boolean variable with the same interval width. Instead of deciding by the interval width this heuristic decides by a measure of how often a variable can still be decided. This is achieved through dividing the interval widths of real variables by the minimal splitting width (MSW) before comparing their widths with those of integer and Boolean variables. If for example the MSW is 0.1, a real valued variable with an interval width of 1.0 can be split as often as an integer variable with a width of 10.
We called this heuristic type "relative-small-interval-first/last", as it does not take into account the absolute width of a variable interval but considers an interval small or great in relation to how often it can still be split. An integer valued variable's interval is thus relatively smaller than the one of a real valued variable with the same absolute width.

3.2.3 Shrunk-interval-first/last (shrunk-fst, shrunk-lst)

With the shrunk-interval-first heuristic we wanted to decide variables first whose intervals have decreased most since the beginning of the algorithm. The quotient by which an interval has shrunk in comparison to its initial width is an indicator for this.
The initial interval widths of all variables are stored at the beginning of the algorithm after a first ICP has been executed. It is important to execute this first ICP before the intervals are stored, because it is completely arbitrary how large the interval widths are set by the problem definition. The problem definition could define the initial interval of a variable x to be $[-100, 100]$. But in the CNF we could also have unit clauses limiting x to a much smaller range, like $(x \geq 3) \wedge (x \leq 7)$. A first ICP will immediately shrink the interval of x to $[3, 7]$, and 4 will be stored as the realistic initial interval width of x instead of 200.
When two variables are compared, both their current interval widths are divided by their respective initial widths. The smaller the resulting quotient is, the more has the interval shrunk and the variable with the smaller quotient is preferred. Shrunk-interval-last works just the other way around, i.e. it prefers variables whose intervals have shrunk the least.

3.3 Heuristics deciding by occurrences in conflict clauses

Before VSIDS emerged as the generally best decision heuristic in Boolean SAT algorithms, there were several different approaches. Some were focused on satisfying as many clauses as possible in order to satisfy the whole CNF as early as possible. Such decision heuristics prefer variables which occur most unresolved clauses. Because by assigning the suitable value to that variable all, these clauses can be satisfied. If, for example, a Boolean variable x occurs in its negative phase $\overline{x}$ in 17 unresolved clauses, these clauses can be satisfied by assigning *false* to x. And if x in occurs in its positive phase x in 14 unresolved clauses,

these can be satisfied by assigning *true* to x. Assigning *false* to x can be considered a better decision because more clauses are satisfied than through assigning *true*.

Other heuristics aimed at generating unit clauses as quickly as possible. A unit clause is an unresolved clause with only one unassigned literal left. All other literals evaluate to *false*. Hence, the remaining unassigned literal has to be satisfied: the unit clause implies the appropriate assignment. If many implications take place, it is advantageous for the solving process, because variables are assigned without branching in the decision tree. Furthermore, it is always through implications that conflicts are identified and backtracks of the solver are triggered. And conflicts also generate conflict clauses, which have proven the most powerful mean to prune big portions of the search space.

In order to generate unit clauses as quickly as possible, it is also necessary to prefer variables occurring in most unresolved clauses. Because all these clauses can be brought one literal closer to being unit by assigning the value, which makes these literals false. If, for example, a Boolean variable y occurs in its negative phase $\overline{y}$ in 23 clauses which are still consistent, all these clauses can be brought one literal closer to being unit by assigning *true* to y. So, the heuristics trying to generate unit clauses work quite similar to those trying to satisfy as many clauses as possible. Both prefer the same variables. The only difference is that the former choose the assignment making most literals *false* whereas the latter choose the assignment making most literals *true*.

An addition to the unit clause generating heuristics can be, that not only the number of occurrences in unresolved clauses is taken into account, but that also the length of these clauses is considered. Because a short clause is closer to becoming unit than a long one. The length of a clause would have to be the number of its unassigned literals.

When VSIDS emerged, it turned out to outperform most of the previous approaches. But this was the case in purely Boolean SAT. In the SAT modulo non-linear interval arithmetic it does not necessarily have to be the same. That is why we decided to adopt several pre-VSIDS approaches in Boolean SAT for iSAT.

The unit clause generating or clause satisfying heuristics just described seem intuitively sensible in their functionality. But adopting them for iSAT is not trivial. They make literals *true* or *false* by only one decision. In Boolean SAT this cannot be any other way because a Boolean variable can only be assigned once with *true* or *false*. In iSAT, however, clauses consist of constraints including integer and real valued variables, which can be split several times. Some constraints include more than one variable and can therefore sometimes not be made *true* or *false* with a single decision. Other constraints include transcendental functions and can therefore not be satisfied either.

The only constraints which can always be satisfied with a single decision are the so called simple bounds. These consist of only a variable, a relation $<, \leq, \geq$ or $>$ and a constant value; for example $(x \geq 2.4)$ or $(y < -4)$. Therefore, the iSAT adoptions of the above heuristics are only considering clauses, which only consist of simple bounds. Such clauses are all conflict clauses (see Section 2.6). That is why the heuristics in this section are all working with the variable occurrences in conflict clauses.

3.3.1 Most occurrences in conflict clauses (moicc)

This heuristic goes by the number of occurrences in conflict clauses. The more occurrences in conflict clauses a variable has, the more it is preferred. Thus, each variable has a counter, which is increased every time it is used in a newly learned conflict clause.

3.3.2 Most occurrences in watched constraints of conflict clauses (moiwccc)

In iSAT every clause has two watched constraints which are the equivalent of watched literals in Boolean DPLL. As long as a clause has two different watched constraints, it is not unit. Watched constraints have to be consistent (the equivalent of an unassigned literal in Boolean DPLL). If the assignment of a variable leads to a watched constraint becoming *false* a new consistent constraint of this clause has to be selected. The moiwccc heuristic counts in how many watched constraints a variable currently occurs and prefers the variables occurring in most.

3.3.3 Most occurrences in shortest conflict clauses (moiscc)

The moiscc heuristic can be considered an iSAT equivalent of Böhm's heuristic. Every variable has counters to indicate in how many conflict clauses of different lengths it occurs. These counters are used to fill the Böhm specific H vectors (see Section 2.4.1). As in the Boolean original of Böhm's heuristic, variables with the smallest H vector in lexicographic order are preferred.

3.3.4 Jeroslow-Wang: Most occurrences in many short conflict clauses (jw)

The Jeroslow-Wang adaptation for iSAT uses the same counters as the moiscc heuristic, which count the occurrences of each variable in conflict clauses of different lengths. These values are used to compute the JW function for every variable (see Section 2.4.2). Variables with the greatest JW value are preferred.
In the Boolean original of Jeroslow-Wang there are two versions: one-sided and two sided (see Section 2.4.2). One-sided differentiates between Boolean literals in positive and negative phase. We do not differentiate between different kinds of simple bounds here. Hence, our adaptation is the equivalent of two-sided Jeroslow-Wang.

3.4 Miscellaneous heuristics

3.4.1 Natural (no heuristic)

The "natural" decision order is not an actual heuristic, because the decision order is not changed. All variables are decided in the order in which they were created during the initialisation of iSAT. Thus, the variables are, however, *not* decided in a random order. It might very well be, that deciding them in the order they were created has its benefits.

3.4.2 VSIDS

The iSAT equivalent of VSIDS works similar to the Boolean original (see Section 2.4.4): variables are rewarded when they occur in conflict clauses. But variables in iSAT have intervals with a lower and upper bound each. When a variable occurs in a new conflict clause, VSIDS increases the activity of either its lower or upper bound, depending on which bound was responsible for the conflict and hence is included in the conflict clause.
This allows VSIDS to differentiate between the activities of a variable's upper and lower bound. When two variables are compared, the maxima of their respective upper and lower bounds are compared. The variable with the higher activity on one of its bounds is preferred for decision.

3.4.3 max-cand, sum-cand

There are two heuristics, "max-cand" and "sum-cand", working with values gathered by the additional "sb-split-cand" option (see Section 4.2). Activating one of these heuristics automatically activates "sb-split-cand". As they are decision heuristics, they are mentioned in this section. But to understand how they work it is necessary to have read about the "sb-split-cand" option first. Hence, "max-cand" and "sum-cand" are explained in Section 4.2.3.

Chapter 4

Additional options

4.1 Resorting options

To understand the resorting options, the reader's attention needs to be turned to a particular aspect of iSAT's implementation. This is the vector object *sorted_vars[]*, in which the pointers to all variables are stored. When iSAT has to decide a variable, it tries the variable whose pointer stands at position *next_var_pos* of *sorted_vars[]*. The integer variable *next_var_pos* is 0 at the beginning of the algorithm. After every attempt[1] to decide a variable, *next_var_pos* is incremented; or set back to 0 when the end of *sorted_vars[]* has been reached.
What decision heuristics in iSAT do is to sort the vector *sorted_vars[]* in a specific manner. Dominant-first, for example, sorts all dominant variables into the first positions of *sorted_vars[]*, such that they are decided first. If another heuristic is activated in combination with dominant-first - for example small-interval-first - the dominant variables are still sorted into positions preceding the auxiliary ones. But the orders among the dominant and among the auxiliary variables are determined by small-interval-first.
A comparison between dominant-first and small-interval-first makes clear why resorting is necessary for some heuristics and for others not. With dominant-first all variables are sorted according to whether they are dominant or auxiliary. These attributes never change during the solving process. Hence, a resorting of *sorted_vars[]* would have no effect on the sorting order. Small-interval-first, on the other hand, sorts the variables according to their current interval width, which is a rather frequently changing variable attribute. Thus, a resort of *sorted_vars[]* is likely to yield different sorting orders at different times during the solving process.
After every resort, *next_var_pos* is reset to 0, because *sorted_vars[0]* holds the pointer to the most preferred variable. If one wishes to always decide the variable which is currently be preferred by the activated heuristics, one has to execute resorts rather often. But this might sometimes not be desired. Consider again the case of small-interval-first: The first variable to be decided is the one with the smallest interval. As a consequence of its decision some other variables' intervals may shrink as well through ICP. But the interval of

[1] A variable does not necessarily have to be decidable at any point of the algorithm. Boolean variables can only be decided once and integer or real valued variables can only be decided as long as their intervals are still bigger than 1 or bigger than the minimal splitting width respectively.

the decided variable has definitely been divided in half. So it is probable that it will again be the variable with the smallest interval. In that case a resort would put the variable again at position 0 of *sorted_vars[]*. And as *next_var_pos* is reset to 0 the same variable would be decided again.
Whether such a repeated decision of only one variable is generally disadvantageous for the overall search process or not is a question which eventually has to be answered empirically. But this example makes clear, why it is not only necessary to resort *sorted_vars[]* but also to be able to determine at what points of the algorithm such resorts take place.

4.1.1 Resort after i decisions (--sortafter=i)

This option determines that a resort is executed after each i decisions of the solver. It can be conceived, that the more decisions have taken place, the more likely it is that the variables in *sorted_vars[]* are no longer in the correct order.

4.1.2 Resort after conflict (--resort-ac)

With this option activated, iSAT executes a resort after every conflict. A resort at these points of the solving process makes sense because conflicts always entail backtracks. And backtracks often undo a lot of assignments, such that the attributes of many variables might have changed, by which a heuristic determines their preferability.

4.1.3 Dynamic resort (--dynamic)

The "dynamic-resort" option is a resorting mechanism intended to execute a resort when it is probably needed. This should keep the overhead for unnecessary resorts at a minimum. Every time a variable at position *next_var_pos* of *sorted_vars[]* is decided, the succeeding variable at position *next_var_pos+1* is checked as well. If these two adjacent variables are still sorted according to the activated decision heuristics, nothing happens. If they are not sorted, however, a resort is executed after this decision.
Thus, no resort is done until two adjacent variables in *sorted_vars[]* are discovered which are not in the right order any more.

4.2 Simple Bound Split Candidate Lists (--sb-split-cand)

In Section 3.3 we described heuristics which prefer variables occurring in most conflict clauses. The reason for this is that conflict clauses exclusively consist of simple bounds, which can always be satisfied or made *false* through a single decision; given they are still consistent, of course. Take for example the simple bound $(x \geq 5.7)$ and let the current interval of x be $[-10, 10]$. If we wish to make the simple bound *false*, we have to reduce the interval of x to a range in which $(x \geq 5.7)$ can no longer be satisfied. This is the case if the upper bound of x becomes strictly less than 5.7. Thus, by reducing the interval of

x to $[-10, 5.7)$, the simple bound becomes *false*. The strict upper bound 5.7. is only the greatest upper bound to achieve this. Any upper bound which is even smaller than 5.7 would make the simple bound *false*, too.

If, on the other hand, we wish to make $(x \geq 5.7)$ *true*, the interval of x has to be reduced to a range in which the simple bound is always satisfied. This is the case if the lower bound of x becomes greater than 5.7. Here, the lower bound does not need to be strict, because 5.7 itself is a satisfying assignment. Thus, by reducing the interval of x to $[5.7, 10]$, the simple bound becomes *true*. But any other lower bound which is even greater than 5.7 would make it *true* as well.

With the ability to make consistent simple bounds *true* or *false* by a single decision, we have an iSAT equivalent of making unassigned literals *true* or *false* in Boolean SAT. Most Boolean heuristics, which inspired us for the heuristics of Section 3.3, count a variable's occurrences separately for its positive and negative phase. Depending on which phase has more occurrences and whether one wishes to make these occurrences *true* or *false*, the variable is assigned. If for example the negative phase $\overline{z}$ of a variable z has more occurrences than its positive phase and the goal is to make many literals *true*, z is assigned *false*.

The iSAT equivalent of counting occurrences in positive and negative phase is to count occurrences in consistent simple *lower* bounds or consistent simple *upper* bounds. Because all consistent simple upper bounds, in which a variable occurs, can be made *true* or *false* with a single decision. And the same is possible for all consistent simple lower bounds. We refer to these consistent simple bounds as split candidates. An example will demonstrates how these split candidates are counted and how the influence the decision of their variable.

Let x be a variable whose interval is $[-10, 10]$ and let x occur in the following simple bounds of conflict clauses:

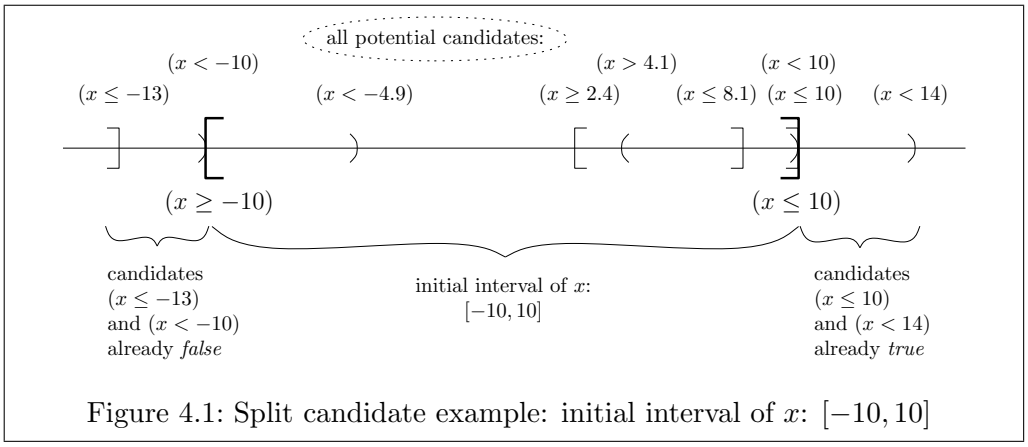

Figure 4.1: Split candidate example: initial interval of x: $[-10, 10]$

$$
\begin{array}{ll}
(x \leq -13) & (1) \\
(x < -10) & (2) \\
(x < -4.9) & (3) \\
(x \leq 8.1) & (4) \\
(x < 10) & (5) \\
(x \leq 10) & (6) \\
(x < 14) & (7) \\
(x \geq 2.4) & (8) \\
(x > 4.1) & (9) \\
(x > 12.8) & (10)
\end{array}
$$

The interval of x and its split candidates are visualised in Figure 4.1. We will go through the candidates one by one and explain if and why they are counted: (1) is already *false* under the variable interval $[-10, 10]$ of x. Because if $x \geq -10$, it cannot be ≤ -13. (2) is also already *false*, because it requires x to be *strictly* smaller than 10. Thus, (1) and (2) are not counted as split candidates.
The values of (3) and (4) are still within the current interval $[-10, 10]$. So, they are consistent and as they are both upper bounds, we count two upper bound split candidates. (5) is a consistent upper bound as well, which brings us to a total of three upper bound candidates. (6) is already *true* under the current interval of x, because all points in $[-10, 10]$ satisfy the requirement $(x \leq 10)$. The same applies for (7), which is an even greater upper bound. Hence, neither (6) nor (7) are counted as upper bound candidates. (8) and (9) are both consistent lower bounds. Hence, we count two lower bound occurrences. (10) is already *false* for the interval $[-10, 10]$, hence not counted.
So, we have three upper bound candidates and two lower bound candidates, which makes the upper bounds more interesting for our heuristics aiming at making simple bounds *true* or *false*. The three simple upper bounds could be made *true* or *false* with only one decision. If we want to make all upper bound candidates *true*, we have to take the smallest upper bound candidate as the new upper bound for the variable interval. Consequently, this smallest upper bound candidate is satisfied. And the other upper bound candidates are satisfied as well, because they have even greater values. In our example with $x \in [-10, 10]$, the three upper bound candidates are: $(x < -4.9)$, $(x \leq 8.1)$ and $(x < 10)$. If we want to make them all *true*, we take the smallest of them $(x < -4.9)$ and make it the new upper bound of the interval of x. This gives us $x \in [-10, -4.9)$ and all our three candidates are satisfied. See Figure 4.2.
If we want to make all upper bound candidates *false*, we have to look at the greatest upper bound candidate and take an even greater value as the new lower bound of the variable interval. Consequently, this greatest upper bound is *false*. And the other upper bound candidates are *false* as well, because they have even smaller values. In our example, we have to take the greatest upper bound candidate $(x < 10)$ and make 10 the new lower bound of the interval of x. This gives us $x \in [10, 10]$ and all three candidates are thus *false*. See Figure 4.3.

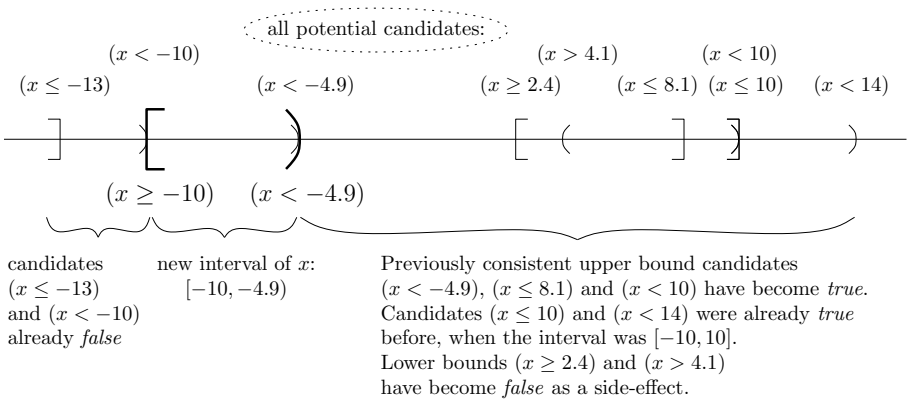

Figure 4.2: Split candidate example: new interval of x: $[-10, -4.9)$ making all upper bound candidates *true*

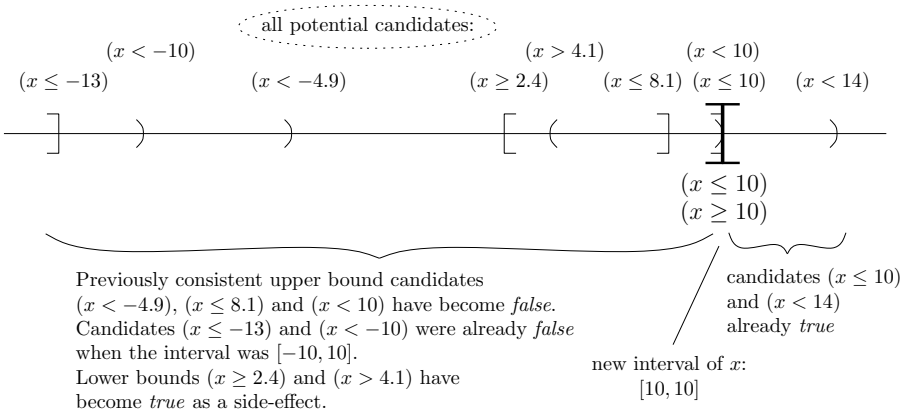

Figure 4.3: Split candidate example: new interval of x: $[10, 10]$ making all upper bound candidates *false*

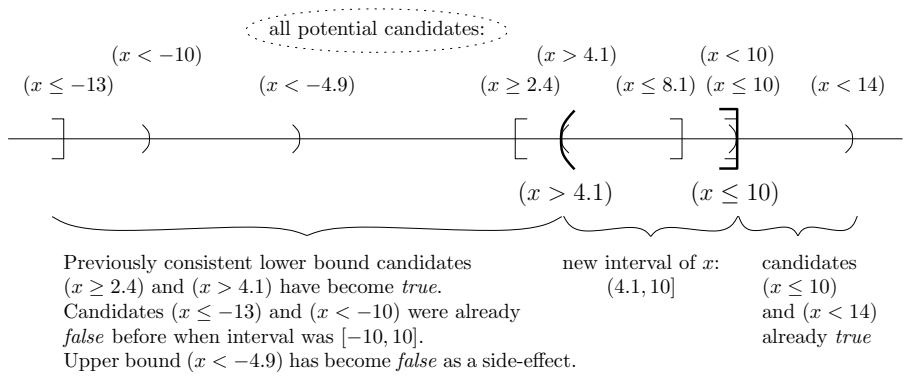

Figure 4.4: Split candidate example: new interval of x: $(4.1, 10]$ making all lower bound candidates *true*

If we want to make all lower bound candidates *true*, we have to take the greatest lower bound candidate as new lower bound for the variable interval. Consequently, this greatest lower bound candidate is satisfied and with it all other lower bound candidates. In our example with $x \in [-10, 10]$ we had $(x \geq 2.4)$ and $(x > 4.1)$ as lower bound candidates. To make them both *true* we have to take 4.1 as the new strict lower bound for the interval of x. With $x \in (4.1, 10]$ both the lower bound candidates are now satisfied. See Figure 4.4.

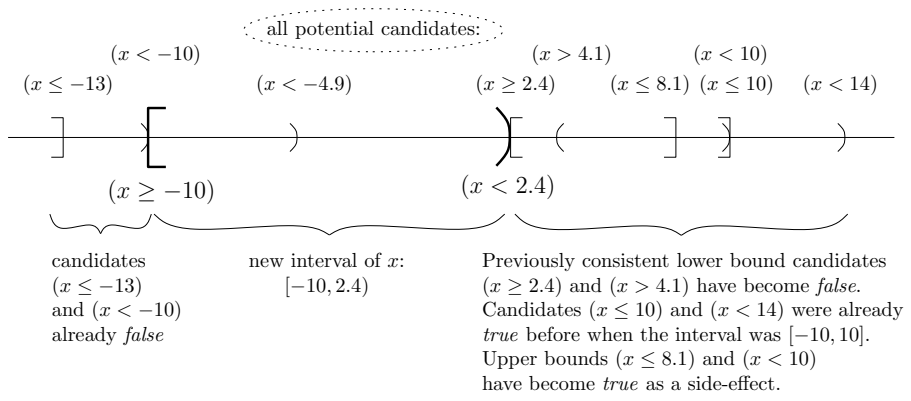

Figure 4.5: Split candidate example: new interval of x: $[-10, 2.4)$ making all lower bound candidates *false*

If we want to make all lower bound candidates *false*, we have to look at the smallest lower bound candidate and take an even smaller value as the new upper bound of the variable interval. Consequently, this smallest lower bound is *false* and with it all other lower bounds. If we want to make the two lower bound candidates, $(x \geq 2.4)$ and $(x > 4.1)$ both *false*, we have to take 2.4 as the new strict upper bound of x. With $x \in [-10, 2.4)$ both $(x \geq 2.4)$

and ($x > 4.1$) are *false*. See Figure 4.5.
The sb-split-cand option is not a decision heuristic itself, because it has no impact on which variables are decided. But it is possible to use their candidate counters to derive heuristics (see below). It rather dictates the value at which a variable is split when decided. The default method is overridden, by which a variable interval is always split in the middle. It also overrides the default method to decide which side of a split interval is considered for the continuing search process.
To determine the split points, it is necessary to store the constants of all simple bound candidates and to store whether these come from a strict or non-strict simple bound. To achieve this, following function is defined:

get_candidate (**Variable** x,
Boolean *make_most_false*)
= (**Boolean** *has_candidate*,
Real *candidate_value*,
Boolean *candidate_is_upper_bound*,
Boolean *candidate_is_strict*)

With:

- the argument x being the variable for whose interval we want to get the new split value,
- the argument *make_most_false* being a Boolean indicating whether it is desired to make most simple bounds *true* or *false* (If most simple bounds shall be made *true*, the returned value has to be either the largest lower bound candidate or the smallest upper bound candidate. If most simple bounds shall be made *false*, it has to be either the smallest lower bound or the greatest upper bound.),
- the return value *has_candidate* being a Boolean indicating if a candidate has been found at all,
- the return value *candidate_value* being the constant from the returned simple bound candidate, which will determine the point at which the interval of x will be split,
- the return value *candidate_is_upper_bound* being a Boolean indicating whether the returned simple bound candidate is an upper or lower bound, and
- the return value *candidate_is_strict* being a Boolean indicating whether the returned simple bound candidate is strict or not.

The four return values allow iSAT to determine the point at which the interval of x needs to be split with respect to whether *make_most_false* was set to *true*. If *make_most_false* was, for example, set to *false* and a non-strict upper bound is returned, iSAT knows that

it is the greatest upper bound candidate. Because by making the greatest upper bound candidate *false*, all smaller upper bound candidates are made *false* as well. Hence, we have to make the returned *candidate_value* the new *strict* lower bound of x. It has to be strict, such that the non-strict greatest upper bound candidate, from which the split value came, is *false* as well.
When *get_candidate* returns an upper bound instead of a lower bound, it is because there were more upper bound candidates than lower bound candidates; and vice versa. The evaluation, which simple bound occurrences of x are still consistent in the current interval of x, is done by *get_candidate*. It also does the comparison between the amounts of upper bound and lower bound candidates of a variable.

4.2.1 --sb-cand-true

In Section 3.3 we already explained the difference between the heuristic types trying to make most constraints *true* and those trying to most *false*. The intuition behind the former was that satisfying constraints will always satisfy the whole clause in which they occur. Hence, satisfying constraints can be expected to bring the algorithm closer to a satisfying assignment for the whole CNF. The intuition behind the latter was that making constraints *false* will bring their clauses closer to becoming unit clauses.
Both alternatives can be conceived to be theoretically justified. For our experiments, we had to be able to try both. The sb-cand-true option enables us to switch between them. By default, sb-split-cand tries to make as many constraints as possible *false* in order to quickly create unit clauses. I.e. the *make_most_false* argument of the abovementioned function *get_candidate*() is set to *true*. By enabling sb-cand-true, *make_most_false* is set to *false*, such that sb-split-cand tries to make as many constraints as possible *true*.

4.2.2 --zero-cand

Enabling the zero-cand option has the effect that 0 is treated as if it was both an upper and lower bound candidate for all variables. The motivation for this is, that iSAT often has to deal with symmetrical functions like $sin(x)$, $cos(x)$ or triplets of the form $y = x^2$. For such functions, everything concerning the evaluation for $x \leq 0$ will have an equivalent in the evaluation for $x \geq 0$. Hence, it can be expected that splitting the interval of x at the point 0 will result in two interval halves for which it makes sense to examine them separately.

4.2.3 Candidate split list specific heuristics

max-cand This heuristic prefers variables for decision which currently have the maximum number of upper bound or lower bound candidates. Let, for example, the variable x have 13 upper bound candidates and 7 lower bound candidates. And let the variable y have 3 upper bound and 16 lower bound candidates. Then, max-cand prefers y as it is possible to make 16 simple bounds *true* or *false* compared to only 13 with x.

Max-cand can be considered an equivalent to the Boolean decision heuristic DLIS (see Section 2.4). DLIS stands for "dynamic largest individual sum (of literals)" and "individual" means that the occurrences are counted individually for positive and in negative phase. This is similar to what max-cand does, because occurrences in upper bounds and lower bounds are counted individually.

sum-cand This heuristic prefers variable for decision which currently have most split candidates in total. This value is calculated by adding the amount of upper bound candidates and lower bound candidates. Let x be a variable with 13 upper bound candidates and 7 lower bound candidates. And let the variable y have 3 upper bound and 16 lower bound candidates. sum-cand adds up these candidate counts to totals of $13 + 7 = 20$ for x and $3 + 16 = 19$ for y. As x has the higher sum, it is preferred.
sum-cand can be seen as an equivalent to the Boolean decision heuristic DLCS (see Section 2.4). DLCS stands for "dynamic largest combined sum (of literals)" and "combined" means that the occurrences in positive and in negative phase are added. This is the same principle followed by sum-cand, as the occurrences in upper bounds and lower bounds are added as well.

4.3 Ignore true or implied clauses (no-ti)

The heuristics described in Section 3.3 and the option sb-split-cand (see Section 4.2) all work with variable occurrences in conflict clauses and are, by different means, aiming at making these conflict clauses *true* or making most of their contained constraints *false*. Now, the question can arise: Why do we still want to reward variables and handle them as more preferable to decide, if most of their occurrences we counted are actually in conflict clauses which are already *true*?
If a constraint is still consistent and belongs to a clause which is already *true* due to another of its constraints being *true*, a decision to make the consistent constraint *true* will not change the status of the clause. Just as little will a decision to make the consistent constraint *false* bring the clause closer to being a unit clause, because if the clause is already satisfied it cannot become unit any more.
Hence, it can be considered desirable to ignore occurrences in conflict clauses, which are already *true*, just as to ignore split candidates belonging to *true* conflict clauses. This is exactly the effect of activating the no-ti option. Furthermore, it can be considered just as idle as counting occurrences in *true* clauses, to count occurrences in *implied* clauses. Through the activation of no-ti, implied clauses are ignored in the same way as *true* clauses.

4.4 Ignore false constraints (false-sb)

In Section 4.3 we explained, why it can make sense that constraints from *true* or implied clauses are ignored by the heuristics working with variable occurrences in conflict clauses

and by the sb-split-cand option. Similarly, it can be argued that the occurrence of a variable in a conflict clause should not be counted by the heuristics described in Section 3.3, if the constraint it occurs in is already *false*, because the simple bound constraint will remain *false* after the decision of its variable and hence not entail any immediate progress in the search process.
The false-sb option has the effect, that occurrences in *false* simple bounds are not counted by these heuristics. Activating false-sb has no effect on the sb-split-cand option, as the latter only counts simple bounds as candidates if they are still consistent. Hence, *false* simple bounds are ignored by it anyway. Neither, has false-sb an effect on the heuristic moiwccc, because a *false* constraint can never be a watched constraint. Hence, *false* simple bounds are ignored by that heuristic anyway, as well.

4.5 Strict (strict)

The option strict has an effect on whether the value of *next_var_pos* is incremented after a decision. In Section 4.1 we explained how *next_var_pos* is the integer number indicating the position in the vector *sorted_vars*, at which the pointer to the next variable to decide is stored. Normally, *next_var_pos* is incremented after every decision and only reset to 0 after a resort of *sorted_vars*. But this might not always be what we want.
Take for example the case in which the active heuristic is small-interval-last. This means that variables with the greatest interval are preferred for decision. Then let the first entries of *sorted_vars* be the variables x, y, z with the following intervals widths:

$$\begin{aligned} &\textit{sorted\_vars}[0] = \text{pointer to } x && \text{(interval width} = 28) \\ &\textit{sorted\_vars}[1] = \text{pointer to } y && \text{(interval width} = 10) \\ &\textit{sorted\_vars}[2] = \text{pointer to } z && \text{(interval width} = 8) \end{aligned}$$

The first variable to decide is x at position 0 of *sorted_vars*. In course of the decision, the interval of x is split in the middle, such that the new interval width becomes $28/2 = 14$. After the decision *next_var_pos* is incremented and y at position 1 of *sorted_vars* is decided. But this can be considered not to be, what the heuristic should actually do, because even after the decision of x its interval with a width of 14 is still greater than the interval of y with a width of only 10.
A way to circumvent this would be to activate the heuristic in combination with the resorting option sortafter=1 (see Section 4.1.1). This will resort the vector *sorted_vars* after every decision and reset *next_var_pos* to 0. Hence, we can be sure that at each decision the variable with, in the case of small-interval-last, currently greatest interval is decided. Resorting, however, can become quite consumptive in terms of computing time, such that the strict option embodies a cheap alternative.
What strict does is that after every decision, the solver compares the variable which has just been decided and whose pointer is stored at position *next_var_pos* of *sorted_vars* with the succeeding variable at position *next_var_pos*+1. Normally, *next_var_pos* is incremented

after every decision. But with strict activated, this incrementation only takes place, if the compared variables (i.e. the one just decided and its successor) are *not* ordered any more according to the respective decision heuristic.
So, consider once more the aforementioned example: After x at position 0 of *sorted_vars* has been decided, its interval has shrunk from 28 to 14. Normally, *next_var_pos* would be incremented, but with strict activated x is first compared with its successor y at position 1. And as the interval of x is still greater than the interval of y, which is 10, *next_var_pos* is not incremented and x is decided again. After this second decision, the interval of x has been halved again, such that its width is now $14/2 = 7$. When x is now compared to y, it is found that they are *not* ordered according to small interval first, because the interval of x is now smaller than the interval of y. Hence, *next_var_pos* is incremented from 0 to 1 and y is decided next. After the decision of y its interval has shrunk to $10/2 = 5$, which is already smaller than the interval of z with a width of 8. Hence, *next_var_pos* is incremented immediately after the first decision of y and z is decided next.
The strict option can be activated with any heuristic or combination of heuristics. It will always have the effect that the variable just decided is compared to its successor, and only if the two are *not* ordered according to the heuristic any more, the successor variable is considered for decision next.
Thus, if strict is, for example, activated in combination with small-interval-first, it can be expected that every variable will be decided over and over again until its interval has reached the minimal splitting width, before the next variable in *sorted_vars* is considered for decision. This is because at the beginning every variable at a position *sorted_vars*$[i]$ has already an interval width smaller or equal to the interval width of its successor at position *sorted_vars*$[i+1]$, $i \in \mathbb{N}_0$. And after a variable at *sorted_vars*$[i]$ has been decided, i.e. its interval has been halved, it is very likely that its new halved interval is still smaller than the one of its successor at *sorted_vars*$[i+1]$. Hence, it will be decided again and again. It could only happen through a rather unlikely ICP implication that the interval of *sorted_vars*$[i+1]$ shrinks more than the interval of *sorted_vars*$[i]$, when the latter is decided.

4.6 Pre-Minimal Splitting Width

The minimal splitting width (MSW) in iSAT defines the value down to which the interval of a variable can be split, before it is not considered a decidable variable any more. If MSW is less than 1, all integer variables can be decided exhaustively because if an integer variable with the interval width 1 is split, it immediately becomes a point interval of width 0. This is normally the case, such that MSW only has an effect on real valued variables.
If the interval width of a variable has reached the minimal splitting width, it will not be decided any more and the next variable in the decision order according to the current heuristic is considered for decision. If all variable intervals have reached the minimal splitting width, the search algorithm terminates and returns these intervals as candidate solutions.

The Pre-Minimal Splitting Width option allows us to define another value *pre-msw*, which is greater than MSW but will be handled by the solver just like the actual MSW value. Thus, all variables will through decisions, at first, only be split down to intervals of widths not greater than *pre-msw*. Once, no variables are decidable any more with this *pre-msw* value, a new *pre-msw* value will be calculated by dividing *pre-msw* by another value *pre-msw-div*, which also has to be defined with the activation of the Pre-Minimal Splitting Width option. The effect is, that all variable intervals are shrinking in a rather evenly manner, similar to what the heuristic small-interval-last is aiming at.
Let for example be x, y and z be variables with the following interval widths:

$$width(x) = 28, \qquad width(y) = 10, \qquad width(z) = 8$$

Let furthermore be MSW = 0.1, *pre-msw* initiated with 16, and *pre-msw-div* = 4.
Let us for simplicity reasons assume that there are no ICP implications possible between the three variables. Then what happens is, that at first x is decided, which halves its interval, such that we get the following interval widths:

$$width(x) = 14, \qquad width(y) = 10, \qquad width(z) = 8$$

At this points all interval widths have reached *pre-msw*. Hence, *pre-msw* is divided by *pre-msw-div*, which was 4, giving us the new value *pre-msw* = 4. This allows x to be decided two more times (reducing its interval width from 14 to 7, then from 7 to 3.5) and y as well (reducing its interval width from 10 to 5, then from 5 to 2.5). The interval of z can only be split once (from 8 to 4), before it reaches the value of *pre-msw* again:

$$width(x) = 3.5, \qquad width(y) = 2.5, \qquad width(z) = 4$$

Now, *pre-msw* is again divided by *pre-msw-div*, which gives us *pre-msw* = 1 and allows for two more decisions of x, y and z each:

$$width(x) = 0.875, \qquad width(y) = 0.625, \qquad width(z) = 1$$

Then, *pre-msw* is again divided by *pre-msw-div*, giving us *pre-msw* = 0.25, allowing again for two more decisions of x, y and z each:

$$width(x) = 0.21875, \qquad width(y) = 0.15625, \qquad width(z) = 0.25$$

When iSAT tries to divide *pre-msw* by *pre-msw-div* again, it realises that the new value *pre-msw* = 0.0625 is already less than the value of the actual minimal splitting width $MSW = 0.1$. Hence, the value of MSW is used, which allows for two more decisions of x, y and z each.

$$width(x) = 0.0546875, \qquad width(y) = 0.078125, \qquad width(z) = 0.0625$$

Now, all variable intervals have reached the minimal splitting width and a candidate solution will be returned by the solver.
As backtracks undo previous decisions and thereby restore variable intervals to their previous widths, we also had to implement a functionality which stores the *pre-msw* value for each decision level and restores it in case of a backtrack.

Chapter 5

Experiments

5.1 Procedures and Methodology

After the developing and implementation phase was completed, the new heuristics had to be tested in order to be compared regarding their usefulness. It is understood that there is a measureless amount of possible heuristic and parameter combinations. We wanted to examine each heuristic in its pure form without any additional options. But the ones where resorting could be expected to have an effect we wanted to test as well with the resorting options *dynamic, resort-ac* and three different values (1, 15, and 40) for the *sortafter* option.
Furthermore, the *strict* option (see Section 4.5) should be tried once in combination with each heuristic, as well as two different settings for the *pre-msw* option (see Section 4.6): both settings had the initial *pre-msw* value set to 16, only in one setting *pre-msw-div* was set to 2 and in the other to 4. Then, the *sb-split-cand* option (see Section 4.2) should also be applied in combination with all heuristics, which included 8 different settings of its own:

1. *sb-split-cand* on its own
   (trying to make most constraints *false*).
2. *sb-split-cand* with *sb-cand-true* activated
   (trying to make most constraints *true*).

3.-4. the two points above with *zero-cand* activated.

5.-8. the four points above with *no-ti* and/or *false-sb* activated.
   (We did not test the effects of *no-ti* and *false-sb* separately, but used them in combination only. For some heuristics, like for example *"most occurrences in watched constraints of conflict clauses"* (see Section 3.3.2), *false-sb* has no effect anyway, such that only *no-ti* had to be activated.)

In the end, we still wanted to have time left to run tests where heuristics were combined among each other which performed well in the first test runs individually; for example *boolean-first* combined with *small-interval-first*. Thus, we ended up with over 400 different settings, which we found worthy of running tests for. In order to be able to execute so many tests, we had to limit ourselves to a restricted amount of benchmark instances and a suitable timeout after which the solving process of a benchmark was aborted.

As timeout we chose 200 seconds for each of the 40 benchmark instances, which we selected from the iSAT benchmark repository. These benchmarks consisted of real life BMC problems (i.e. models of discretised hybrid systems) but also of developer benchmarks (i.e. designed test cases from earlier iSAT creation phases). Among these 40 benchmarks were 20, which iSAT could identify as being unsatisfiable within 200 seconds each, when the "natural" decision order was used. For 10 a candidate solution was found. And the remaining 10 could not be solved in 200 seconds.
The machines on which we ran our tests were two identical dual-processor computers, each of which had two identical CPUs, namely an "AMD Opteron 252" running with 2.6 GHz and a cache size of 1024 KB. Each computer had a memory of 16 GB, but we limited iSAT to only use 2 GB per benchmark instance at maximum anyway. Thus, we were able to have four instances of iSAT running simultaneously at all times; one on each of the four identical CPUs.
In the following, we will present our findings which we found to be most striking and interesting. It would go beyond the scope of this thesis to include all results, as the gathered data is quite extensive. A detailed description of the individual benchmark files is also omitted, but both the complete test data as the benchmark files are available in the iSAT repository and will be gladly provided on demand.

5.2 Comparison by overall time and other values

In the evaluation phase we gathered data for each individual benchmark and each heuristic/options combination. Among this data were characteristic values such as "maximum decision level depth", "maximum number of decisions per decision level", "number of conflicts", "average conflict clause size", "maximum backjump distance", and "number of conflict clauses implied by conflict clauses". Furthermore, the overall computation time for each benchmark instance was assessed. We measured the times individually which were spent for resorting the *sorted_vars* vector (see Section 4.1) and for maintaining the different efforts of sb-split-cand, no-ti and false-sb (see Sections 4.2, 4.3, and 4.4). The same was done for the extra efforts needed for all heuristics depending on variable occurrences in conflict clauses (see Section 3.3).
To compare the heuristics and their combinations with one another, we added the values of all 40 benchmark instances for each heuristic and compared only the sums. If a benchmark could not be solved within the timeout of 200 seconds, the value 200 was added to this sum of overall computation times. We found that this sum was the most interesting and significant value to estimate the usefulness of a heuristic. Because it could be seen that the heuristics with the smallest overall time sums were also the ones which solved most benchmarks.
Table 5.1, 5.2 and 5.3 show the 20 best heuristic combinations regarding the overall computation times as well as the "natural" decision order setting for comparison. The times spent for resorting *sorted_vars* were all negligible for these heuristics (less than 1 second

for all 40 benchmarks per heuristic) and the same can be said about the time needed for maintaining sb-split-cand (less than 8 seconds). The heuristic combinations in which no-ti was activated (table rows 3, 4, 7, 8, 9, 10, 11, 12, and 18) showed that the maintenance of no-ti is more demanding but still not noteworthy: only 2% of the overall computation time was needed for no-ti operations.

	Heuristic combination	Overall time [s]	#solved of 40	#unknown	#unsat	#cc implied by cc
1	shrunk-interval-first boolean-first strict sb-split-cand sb-cand-true	1465.46	36	12	24	9681
2	boolean-first shrunk-interval-first strict sb-split-cand sb-cand-true	1471.50	36	12	24	9681
3	boolean-first shrunk-interval-first strict sb-split-cand sb-cand-true no-ti	1548.89	36	13	23	7209
4	shrunk-interval-first boolean-first strict sb-split-cand sb-cand-true no-ti	1554.45	36	13	23	7209
	natural	2703.04	30	10	20	25100

Table 5.1: The 4 best heuristic combinations regarding the sum of all 40 benchmarks' overall computation times. Values of "natural" (no heuristic) are given for comparison.

The false-sb option does not appear among the best 60 heuristics, but then again it only makes sense in combination with certain heuristics, which are not ignoring false simple bounds anyway. It first appears at position 79 in combination with "Jeroslow-Wang sortafter=40 sb-split-cand sb-cand-true no-ti" solving 31 benchmarks in 2489.15 seconds with the time needed for false-sb operations being less than 1% of the overall computation time.
It was not possible to find any patterns in the examination of the abovementioned values "maximum decision level depth", "number of conflicts" etc. An exception can be seen in the value "number of conflict clauses implied by conflict clauses", which is therefore given in the tables under the abbreviation "#cc implied by cc". Apparently there is a correlation between the activation of sb-split-cand with sb-cand-true and a small value of "#cc implied by cc". Even though our four best heuristic combinations in Table 5.1 could

	Heuristic combination	Overall time [s]	#solved of 40	#unknown	#unsat	#cc implied by cc
5	small-interval-first boolean-last sb-split-cand sb-cand-true	1635.06	34	12	22	3195
6	small-interval-first sb-split-cand sb-cand-true	1713.98	34	12	22	3728
7	integer-last boolean-last sb-split-cand sb-cand-true no-ti	1812.09	34	11	23	4807
8	dominant-first boolean-last sb-split-cand sb-cand-true no-ti	1861.27	33	11	22	3501
9	small-interval-first boolean-last sb-split-cand sb-cand-true no-ti	1861.40	33	11	22	3917
10	boolean-last integer-last sb-split-cand sb-cand-true no-ti	1891.24	34	11	23	3877
11	boolean-last sb-split-cand sb-cand-true no-ti	1911.34	33	11	22	3750
12	boolean-last dominant-first sb-split-cand sb-cand-true no-ti	2009.54	34	11	23	4135
	natural	2703.04	30	10	20	25100

Table 5.2: Heuristic combinations on position 5 to 12 of our ranking regarding the sum of all 40 benchmarks' overall computation times. Values of "natural" (no heuristic) are given for comparison.

	Heuristic combination	Overall time [s]	#solved of 40	#unknown	#unsat	#cc implied by cc
13	small-interval-first boolean-last	2043.46	32	10	22	18175
14	integer-last boolean-last sb-split-cand sb-cand-true	2078.99	33	11	22	4329
15	dominant-first boolean-last sb-split-cand sb-cand-true	2107.54	32	10	22	3004
16	boolean-last sb-split-cand sb-cand-true	2122.06	33	11	22	4063
17	only-and-all-dominant-first sb-split-cand sb-cand-true	2132.49	33	11	22	5254
18	small-interval-first sb-split-cand sb-cand-true no-ti	2135.25	33	11	22	4396
19	dominant-first boolean-last	2153.33	35	11	24	26319
20	small-interval-last sb-split-cand sb-cand-true	2154.64	32	10	22	3464
	natural	2703.04	30	10	20	25100

Table 5.3: Heuristic combinations on position 13 to 20 of our ranking regarding the sum of all 40 benchmarks' overall computation times. Values of "natural" (no heuristic) are given for comparison.

be considered outliers of this putative regularity with comparatively great "#cc implied by cc"-values.

Striking is the fact that the sb-split-cand option with sb-cand-true is occurring in almost all heuristics of our computation time top twenty. This gives reason to believe that it does indeed have a positive effect, when variable intervals are split at points such that most simple bounds belonging to learned conflict clauses become *true*. It is furthermore a positive result, that the computation time could be reduced by almost 50% compared to the "natural" decision order and that up to 6 more benchmark instances could be solved.

The four best heuristics (see Table 5.1) are basically one and the same combination of heuristics. The only difference between rows 1 and 2 and between rows 3 and 4 respectively is that in 1 and 4 the variable order is first determined according to shrunk-interval-first and then according to boolean-first. In rows 2 and 3 it is the other way around. The combinations of row 3 and 4 have the no-ti option activated whereas the ones in row 1 and 2 do not.

That boolean-first performed well can be seen as a confirmation of the intuition which led to its creation. It was, however, more of a surprise that shrunk-interval-first with the strict option performed so well. Because shrunk-interval-first without any resorting options does effectively nothing: In the beginning of the solving process, when the initial interval widths are stored, all variables are treated equally by this heuristic, because no variable intervals have shrunk yet. When the strict option is activated, the following happens: After the first variable has been decided, its interval has shrunk and probably more than (through ICP) the interval of its successor variable in *sorted_vars*. Hence, the variable just decided and its successor are still in correct order according to shrunk-interval-first and the just decided variable is decided again. This is very likely to go on until the variable's interval cannot be split any more. Apparently, this is a beneficial method to solve many of the instances from our benchmark set.

Rows 1 to 4 show us, that it does not make much of a difference, which of the two heuristics shrunk-interval-first or boolean-first is taken as primary and which as secondary sorting criterion. The activation of no-ti does make a slight difference though, which is not so much noticed in the slightly longer computation time. But there are two benchmarks, one of which produced a timeout with no-ti but could be solved as *unsat* without no-ti in 16.72 seconds[1], while the other one produced a timeout without no-ti and could be solved as *unknown* (candidate solution) with no-ti in 1.5 seconds[2]. Phenomena like this demonstrate very blatantly what kinds of irregularities and unpredictabilities we are dealing with in such a heuristic research field.

Another heuristic with good results was small-interval-first (rows 5, 6, 9, 13, and 18) without any resorting activated. This was rather unexpected, because small-interval-first without resorting determines the decision order in the very beginning of the solving process

[1] /imira/benchmarks/sisat/process_wireless_ctrl/process_wireless_ctrl_04_sol_nonprob.hys (BMC-depth 30)

[2] /imira/benchmarks/hysat/tomlin_aircraft_roundabout_maneuver_modified.hys (BMC-depth 3)

after a first deduction has taken place. So apparently the initial interval width of a variable does have a significance for its decision preferability. It can be assumed, though, that the effect of small-interval-first without resorting is quite similar to boolean-first. Because Boolean variables have rather small intervals (namely of size 1) by nature. On the other hand, small-interval-last, which is the opposite of small-interval-first also made it into the top 20 table in the last row. This makes it difficult to draw any conclusion in favour of small-interval-first.
Dominant-first (rows 8, 12, 15, and 19) and only-and-all-dominant-first (row 17) are among the 20 best heuristics. But it has to be pointed out, that row 8, 12, and 15 are effectively the same heuristic combination only with a different sorting order priority and no-ti being omitted in row 15. The occurrences of dominant-first appear to be a justification of our intuition. On the other hand, we find dominant-last in combination with sb-split-cand and sb-cand-true at position 62 of our heuristic ranking, solving 31 benchmarks in 2405.13 seconds. So, it might not be justified to argue in favour of dominant-first too enthusiastically. Similarly, the combination of boolean-first with BMC-backward occurring in row 25 seems not enough to claim a major achievement with BMC-backward, but it is noteworthy never the less.
What can definitely be drawn from these results, is the usefulness of sb-split-cand with sb-cand-true. We see this as an indicator for the necessity of examining the actual structure of a given problem formula in order to solve it successfully.

5.3 Correlations between heuristics and between benchmark instances

In an attempt to find out more about the iSAT heuristic problem, we set out to calculate the correlations between all heuristic combinations with one another. The idea was to find out which pairs of heuristics solved the same benchmarks in rather short periods of time. This would enable us to identify, so to speak, "families" of heuristics and we could try what happens if we combine heuristics belonging to the same family or to different uncorrelated families. Likewise, we were interested in whether there are families of benchmarks, which are all solved by about the same sets of heuristics. If a manageable small amount of such benchmark families could be identified, it would be possible to select an equally small set of heuristics, from which it could be expected to solve any new benchmark.[3]
But among our over 400 heuristic combinations there were of course many which did not perform well at all for our 40 benchmarks, i.e. they produced timeouts for most instances. These underachieving heuristics would all have rather high correlations among each other and would also blow up the amount of data to be reprocessed. Hence, we reduced the data, such that we only registered the cases in which a heuristic solved a benchmark in less than

[3]It needs to be said in advance, that the approach presented in this section did not yield the results we hoped for. Never the less, the method and results are presented here as some considerable scientific effort has been dedicated to them. Such endeavour is justified to be regarded in a Diploma thesis.

10 seconds[4]. Effectively this is the same as if we had run all our tests with a timeout of 10 seconds instead of 200.
From this data we created a table, whose rows are the heuristic combinations and whose columns are the benchmark instances. A "-1" at the crossing of a heuristic row and a benchmark column has the meaning that this benchmark was *not* solved by this heuristic in less than 10 seconds; a "1" stands for that it was. Table 5.4 shows a small example of this concept, which we refer to as "coverage table", because the table show which benchmarks are covered by which heuristic.

	ae_test_abs_3.hys	ae_test_min_3.hys	bouncing_ball_euler.hys 57
natural	-1	1	1
BMC-forward	-1	1	-1
VSIDS	1	1	-1

Table 5.4: An exemplary extract of our coverage table, showing that ae_test_min_3.hys is solved by VSIDS, that bouncing_ball_euler.hys with BMC-depth 57 is solved by natural, and that ae_test_min_3.hys is solved by natural, BMC-forward, and VSIDS.

From this coverage table we proceeded to calculating the correlations between all heuristics and between all benchmarks as follows: Let the coverage table be represented as the m-by-n matrix T, whose m rows are the heuristics and whose n columns are the benchmarks. Thus, the entry $t_{j,k}$ (row j, column k) is "1", if heuristic j solved benchmark k in less than 10 seconds; otherwise it is "-1". The correlation between two heuristics, which stand in row i and row j, is then calculated according to the formula:

$$HeuCor(i,j) = \frac{\sum_{k=1}^{n}(t_{i,k} * t_{j,k})}{n}$$

with n being the number of benchmarks. Thus, the more identical two heuristics are regarding which benchmarks they solved or did not solve, the closer is the value of their correlation to 1. And the more they differ regarding their solved benchmarks, the closer is their correlation to -1. The correlation between benchmark instances is calculated analogously:

$$BenCor(i,j) = \frac{\sum_{k=1}^{m}(t_{k,i} * t_{k.j})}{m}$$

with m being the number of heuristics.

[4]The choice of 10 seconds was admittedly arbitrary and could have been any other rather small value. But 10 seconds seemed like a suitable choice, as for all but one benchmark of the 40 we tested, there were at least more than one heuristics which solved it in less than 10 seconds. The remaining benchmark (/imira/benchmarks/hysat/train_distance_ctrl1.hys with BMC-depth 21) produced a timeout with all heuristics.

As a result we got one m-by-m matrix showing the correlations between all benchmarks and one n-by-n matrix showing the correlation between all heuristics. An exemplary extract from the benchmark correlation matrix is shown in table 5.5.

	bug.hys 12	freezer.hys 26	freezer.hys 27	h3_train_jun_isat.hys 17
bug.hys 12	1.00	−0.57	−0.37	−0.78
freezer.hys 26		1.00	0.72	0.80
freezer.hys 27			1.00	0.59
h3_train_jun_isat.hys 17				1.00

Table 5.5: An exemplary extract from the matrix representing the correlations between benchmark instances. The correlation of 0.80 between freezer.hys with BMC-depth 26 and h3_train_jun_isat.hys with BMC-depth 17, for example, indicates that these two benchmarks are solved by a similar set of heuristics. A low correlation, on the other hand indicates that two benchmarks are rarely solved by the same heuristic.

Next, we were interested in finding the abovementioned putative "families" of benchmarks or heuristics. To have a clear result, we did not want to allow a benchmark or heuristic to be the member of more than one family. This, however, confronted us with the problem of how to define a family. The original intuition was that all members of a family should have great correlations with one another. But consider the case where a member m_1 has a great correlation with another member m_2 and another member m_3, but m_2 and m_3 have a very small or even negative correlation between each other. Following the intuition, m_2 and m_3 should not be in the same family but then we would have to decide which family we assign m_1 to.

To tackle this dilemma, it was decided to define a family such that in two different families f_x and f_y there are never two members, one of which belongs to f_x and the other to f_y, which have a correlation between each other above a certain value. Or in other words: If two heuristics or benchmarks belong to different families, they definitely have a correlation smaller than a certain value. This value was implemented as a variable parameter. When set to 1.0, each heuristic or benchmark gets assigned its own family, because there was never a correlation of 1.0 between two different heuristics or benchmarks. When the parameter is set to a sufficiently small value, say 0.0, all heuristics or benchmarks get assigned to the same family, because for every heuristic or benchmark there is at least one other heuristic or benchmark with which it has a correlation greater than 0.0, such that their families are merged.

We experimented with different values for this parameter until a manageable amount of families was returned. But the results were somewhat sobering. Having set the parameter to 0.8, we got 23 benchmark families all but four consisting of only one benchmark. Among the four multi-member families were two, which consisted of two benchmarks each, one consisting of four benchmarks and the remaining family comprised all remaining bench-

marks:

benchmark families 1-19:	*all consisting of only one benchmark*
benchmark family 20:	parking_lin.hys, BMC-depth 43
	parking_lin2.hys, BMC-depth 43
benchmark family 21:	parking_rad_mod_1.hys, BMC-depth 52
	parking_rad_mod_1.hys, BMC-depth 53
benchmark family 22:	bug.hys, BMC-depth 11
	bug.hys, BMC-depth 12
	sample_bmc_lha.hys, BMC-depth 8
	sample_bmc_lha.hys, BMC-depth 9
benchmark family 23:	*comprising the remaining 13 benchmarks*

Family 20 comprised the benchmarks parking_lin.hys and parking_lin2.hys both with BMC-depth 43. This is not surprising, as these two benchmarks are very similar, such that it could be expected that they are solved by similar sets of heuristics. Family 21 comprised the benchmark parking_rad_mod_1.hys with BMC-depth 52 and 53, and family 22 the benchmarks bug.hys with BMC-depth 11 and 12 and sample_bmc_lha.hys with BMC-depth 8 and 9. This means that for these three benchmarks the BMC-depth does not have a great impact on which heuristics are suited best to solve them.
Unfortunately, this cannot be said for all benchmarks, what casts a shadow on the whole idea of identifying benchmark families. When we look at the 19 single-member families, we observe that the same benchmarks with different BMC-depths are assigned to separate families, which means that for these benchmark the BMC-depth does indeed have an effect on which heuristics they are solved by. Thus, we find the benchmark bouncing_ball_euler.hys with BMC-depths 56 and 57 in two different single-member families. The same is the case for the benchmark etcs_train_system_static_removed_mod.hys with BMC-depths 34 and 41 and for highlift4.hys with BMC-depths 64 and 65. Furthermore, we find parking_lin.hys and parking_lin2.hys both with BMC-depth 44 in different single-member families, which has to curb our enthusiasm for having found these two benchmarks with the same BMC-depth in benchmark family 20.
Trying to identify families of heuristics in this way presented a similar scenario. With the parameter set to 0.8, we get 17 families but almost all heuristics are assigned to one of two different gigantic families, which makes it impossible to pick a representative heuristic. The remaining 15 families are all single-membered except two:

heuristic families 1-13:	*all consisting of only one heuristic*
heuristic family 14:	VSIDS sortafter=8 sb-split-cand no-ti
	VSIDS sortafter=40 sb-split-cand
heuristic family 15:	max-cand sortafter=40 sb-split-cand sb-cand-true
	moiwccc sortafter=15 sb-split-cand sb-cand-true
heuristic family 16 and 17:	*comprising the remaining ∼400 heuristics*

This only allows to draw the conclusion that the two heuristics in family 14 and 15 respectively are similar. But then the question is, why it is these parameter combinations and not any other, which were also tested.

Chapter 6

Conclusions and future work

One of the most interesting result of this thesis is that the sb-split-cand option has proven to bring a great advantage for the solving process. With it the solver is able to satisfy many constraints of learned conflict clauses with only one variable decision. This brings the problem of solving non-linear interval constraint formulas closer to the original Boolean SAT domain.
Moreover, it has to be stated that several of the heuristics we tried were able to reduce the overall computation time by almost 50% in comparison with the "natural" decision order. The heuristics which have been implemented provide an extensive ground for future works. One prospect is to implement a portfolio approach, in which a certain set of heuristics is chosen to run in parallel. Also, it can be expected that further insight is gained, when test are run for more than just the 40 benchmarks tried in this work.
The success of sb-split-cand is seen as an indicator, that future research needs to go into a direction where the structure of a benchmark problem is examined more thoroughly. On basis of such information, it would then be possible to estimate a suitable heuristic. Further heuristics can easily be derived from the range of techniques developed for this thesis.

Bibliography

[1] The iSAT web page, available at http://isat.gforge.avacs.org, 2010.

[2] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[3] Roberto J. Bayardo and Robert C. Schrag. Using CSP lock-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 203–208, July 1997.

[4] Daniel Brand. Verification of Large Synthesized Designs. In *IEEE/ACM International Conference on Computer Aided Design*, pages 534–537, 1993.

[5] Michael Buro and Hans Kleine-Büning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, (49):143–151, 1993.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.

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

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

[9] Andreas Eggers, Natalia Kalinnik, Stefan Kupferschmid, and Tino Teige. Challenges in constraint-based analysis of hybrid systems. In *CSCLP*, pages 51–65, 2008.

[10] Martin Fränzle, Christian Herde, Stefan Ratschan, Tobias Schubert, and Tino Teige. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT) – Special Issue on SAT/CP Integration*, 1:209–236, 2007.

[11] Evgueni Goldberg and Yakov Novikov. Berkmin: a fast and robust sat-solver. In *Proceedings of the Design Automation and Test in Europe (DATE 2002)*, pages 142–149, 2002.

[12] John N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[13] Robert J. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–188, 1990.

[14] Henry A. Kautz and Bart Selman. Planning as Satisfiability. In *10th European Conference on Artificial Intelligence*, pages 359–363, 1992.

[15] Henry A. Kautz and Bart Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *13th AAAI National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[16] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the fifteenth International Joint Conference on Artificial Intelligence IJCAI'97*, pages 366–371, Nagayo, Japan, 1997.

[17] João P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *In 9th Portuguese Conference on Artificial Intelligence (EPIA)*, pages 62–74, 1999.

[18] João P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, 1996.

[19] João P. Marques-Silva and Karem A. Sakallah. Boolean Satisfiability in Electronic Design Automation. In *IEEE/ACM Design Automation Conference*, pages 675–680, 2000.

[20] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 530–535, 2001.

[21] Ofer Strichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1):5–24, 2004.

[22] Grigori S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, 2:115–125, 1968.

[23] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE '97), volume 1249 of LNAI*, pages 272–275, 1997.

[24] Lintao Zhang, Connor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 279–285, November 2001.

List of Figures

List of Tables