Technische Universität München

Fakultät für Informatik

Entwicklung eines Werkzeugs zur Lösung gemischt logisch/linearer Constraint-Probleme

Diplomarbeit

Michael Tautschnig

Technische Universität
München

Fakultät für Informatik

Entwicklung eines Werkzeugs zur Lösung gemischt logisch/linearer Constraint-Probleme

Diplomarbeit

Michael Tautschnig

Aufgabensteller:	Prof. Dr. Manfred Broy
Betreuer:	Dipl.-Inform. Andreas Bauer
Abgabedatum:	15. Februar 2006

Erklärung: Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.02.2006

Technische Universität
München

Fakultät für Informatik

Development of a tool to solve mixed logical/linear constraint problems

A thesis submitted for the degree of
Diplom Informatik

Michael Tautschnig

Supervisor:	Prof. Dr. Manfred Broy
Instructor:	Dipl.-Inform. Andreas Bauer
Submission date:	February 15, 2006

Acknowledgements

Many people helped getting this work to its current state and hereby I'd like to thank

- My instructor Andreas Bauer for his valuable advice, answers to all my questions and for many insightful discussions.
- My supervisor Prof. Dr. Manfred Broy for giving me the opportunity to turn this work into a diploma thesis.
- Christian Schallhart for patiently sharing his knowledge of C++ with me.
- Dr. Stefan Katzenbeisser, whose ideas led me to the first contacts with the underlying matter.
- My family and Samira for their everlasting support and all the patience.

Abstract

The problem of solving mixed arithmetic and Boolean constraint systems arises in many different areas, such as verification of soft- and hardware systems, resource planning or system design and has been studied extensively in recent time. Yet, the available solvers are neither easily extensible, nor do they offer ways to apply problem specific heuristics that are required for most of the hard problems in this area.

To overcome these limitations, a framework has been designed to integrate state-of-the-art solvers for the Boolean- and parts of the arithmetic domain to solve the combined problem. Thereby we benefit from the full strength of each of the tools in their special area. Furthermore, the architecture of the system emphasises extensibility, which already proved useful for the implemented extension to non-linear arithmetic constraints.

The results show that our implementation, albeit in in some parts not yet more than a proof of concept, can already compete with existing solvers. Due to the extension to non-linear arithmetic we are even able to tackle a new class of real-world problems.

The present work introduces this class of problems and our approach to solve them, accompanied by some real-life examples. Along with these descriptions we provide detailed insight into our tool and the hurdles we had to overcome.

Contents

Introduction

Once working on the subject of mixed Boolean and arithmetic constraint systems, one is tempted to see these problems everywhere. Still, let us start out with an example to illustrate the type of problem we are to deal with. Consider the following piece of C code, where we added an assertion to ensure that there is no array access beyond its bounds. This is a common cause of illegal memory access, also known as "segmentation fault". Note, that the code is solely meant to demonstrate an application of mixed Boolean and arithmetic systems, thus any other commands were left out.

```c
void fn( int * a, int width, int size )
{
  int i = 0, j = 0;

  for( i = 0; i < size / width; ++i )
    for( j = 0; j <= width; ++j )
      assert( i * width + j < size );
}

int main( int argc, char * argv[] )
{
  int a[ 10 ];

  if( argc > 5 )
    fn( a, 1, 10 );
  else
    fn( a, 2, 10 );

  return 0;
}
```

To decide as to whether the assertion will fail, one has to test for the satisfiability of the following logical combination of arithmetic constraints over integers:

$$
\begin{array}{ll}
 & (i \geq 0) \\
\wedge & (j \geq 0) \\
\wedge & \Big(\big(\neg(i + j < 10) \wedge (i < 10) \wedge (j \leq 1) \big) \\
 & \vee \big(\neg(2i + j < 10) \wedge (i < 5) \wedge (j \leq 2) \big) \Big)
\end{array}
$$

This system can be solved manually and we find $i = 9$, $j = 1$ and $i = 4$, $j = 2$ to be feasible solutions. As any experienced C programmer has spotted already, the "`<=`" in line six should have been a "`<`" instead. That makes the system unsatisfiable and thus the assertion will never fail, as can be proved using the modified system where $\leq$ is replaced by $<$.

The aims of this thesis

The example scratched on the surface of a class of problems occurring in model checking (cf. [Clarke et al., 1999]) of systems over infinite domains, like integers or real numbers. As such, the underlying problem has been studied extensively in recent time and is part of the emerging field of *satisfiability modulo theories* (SMT). First results in this area were provided by LPSAT [Wolfman and Weld, 1999]. Today solvers such as MathSAT [Bozzano et al., 2005] allow for efficient treatment of many instances of mixed Boolean- and *linear* arithmetic problems via tight integration of the logical and arithmetic solver procedures. As a result, they are not easy to extend and even more, they do not deliver the growing power of solvers for the Boolean and various arithmetic domains.

To overcome these limitations, we propose a new Open Source framework that allows the integration of existing Boolean and arithmetic solvers. Thereby we can even enter new domains, as we do using a non-linear solver. As the set of use cases in Chapter 3 on page 23 shows, our approach thus supports new applications of this type of solvers.

Furthermore, the Open Source model features wide spread development and allows other applications to reuse our code. To simplify the latter, our framework may as well be used as a library and can thus be integrated in, e. g. model checkers.

Outline

The present thesis is organised as follows: In Chapter 1 the problem is specified more formally and an introduction to the relevant part of complexity theory is provided. Our approach is presented in Chapter 2, the main experimental results and applications are listed in Chapter 3. We end the main part with our conclusions and some possible extensions in Chapter 4. In the appendix we provide the grammar of our input language and an example session to make the user familiar with our tool.

Chapter 1

Basic concepts

Even though the problem of mixed Boolean and arithmetic constraints occurs so frequently in areas such as verification, there is no known common formal specification. Rather it is redefined by the limitations of each solver, such as the "math-formula" in [Bozzano et al., 2005]. As they only deal with linear arithmetic and the framework explained in Chapter 2 by design allows for any kind of arithmetic or Boolean operation, we are to define the class of problems we can effectively solve using the current system. To simplify the notation, we refer to this class as $\mathcal{AB}$, which is inductively defined in the following paragraphs.

For a set of variables V, let $B(V)$ denote the Boolean formulae over V, built using the operators $\vee$, $\wedge$ and $\neg$. Furthermore, let $\mathbb{B}$ be the set of Boolean constants, i. e. $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$. A substitution is a mapping $\sigma : V \to \mathbb{B}$, which means a replacement of each variable by a Boolean constant, hence the truth value of the formula can be obtained. Given $\sigma : V \to \mathbb{B}$ and $\beta \in B(V)$, we say that σ satisfies β, written as $\sigma \models \beta$, if and only if $\beta(\sigma) = \mathit{true}$. The Boolean expressions in $\mathcal{AB}$ are exactly those of $B(V)$.

The arithmetic expressions in $\mathcal{AB}$ are defined as follows: Let $\mathbb{Q}$ be the set of rational numbers. Any numerical constant $c \in \mathbb{Q}$, as limited by the precision of the effective finite data types, is in $\mathcal{AB}$, as well as any set of arithmetic variables V'. An arithmetic term is built of variables $v' \in V'$ and constants, concatenated by multiplication, division, addition or subtraction. These terms, as well as appropriate parentheses, are in $\mathcal{AB}$. By adding the comparison of arithmetic terms over V' using the operators $\leq, <, \geq, >$ and $=$ we constitute the link to Boolean expressions and denote these comparisons, similar to $B(V)$, by $A(V')$. Furthermore, we can define a substitution as we did for Boolean expressions, such that we can obtain the numerical value of any arithmetic term and the truth value of each comparison in $A(V')$.

To constitute a mixed Boolean and arithmetic system, each variable $v \in V$

may be replaced by an arithmetic comparison from $\mathcal{AB}$ using the mapping $\alpha : V \to A(V')$. For the example from the introduction one would obtain the following terms:

$$\begin{aligned}
\beta &= b_1 \wedge b_2 \wedge \Big((\neg b_3 \wedge b_4 \wedge b_5) \vee (\neg b_6 \wedge b_7 \wedge b_8)\Big) \\
\alpha(b_1) &= i \geq 0 \\
\alpha(b_2) &= j \geq 0 \\
\alpha(b_3) &= i + j < 10 \\
\alpha(b_4) &= i < 10 \\
\alpha(b_5) &= j \leq 1 \\
\alpha(b_6) &= 2i + j < 10 \\
\alpha(b_7) &= i < 5 \\
\alpha(b_8) &= j \leq 2
\end{aligned}$$

This mapping allows for looking at instances of $\mathcal{AB}$ as disjoint sets of Boolean and arithmetic problems, which simplifies the solution process.

Before turning to the actual solvers of this kind of problems, a brief discussion of the computational complexity of deciding instances of $\mathcal{AB}$ is provided in Section 1.4 on page 7. To allow a consistent discussion of this matter we provide a short introduction to Turing machines and the relevant definitions. For a more extensive reference regarding this subject consult, e. g. [J. E. Hopcroft, 2001].

1.1 Complexity theory

Whereas the idea of efficient algorithms was already known to the ancient Greeks [Fortnow and Homer, 2002], the research area of *computational complexity* is still young and roughly dates back to the paper of Hartmanis and Stearns [Hartmanis and Stearns, 1965], where the definitions of time- and space complexity on multitape Turing machines were first laid out.

1.1.1 The Turing machine

An *alphabet* is a finite and nonempty set of symbols, usually denoted by Σ. A *word* is a finite sequence of symbols chosen from some alphabet. It may have a length of 0, in which case it is said to be the *empty word*, denoted by ε. A *language* L is a subset of Σ^*, the set of all words over an alphabet Σ.

A *Turing machine* M is constituted of a *finite control* and an infinite *tape* of *cells*, which can hold a symbol as specified below, and a *tape head*

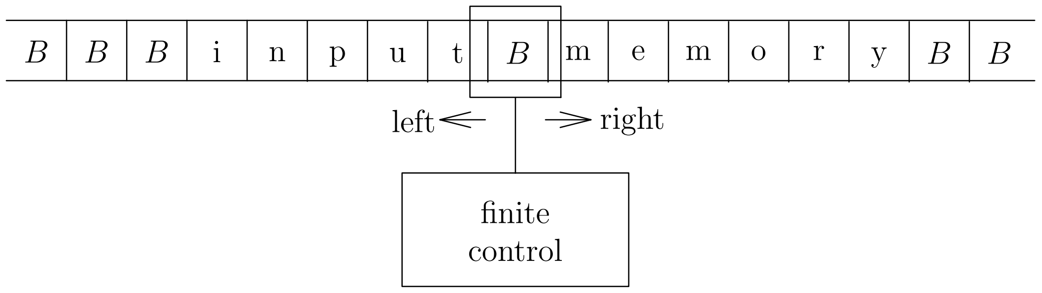

Figure 1.1: A Turing machine

moving over the tape, which *scans* the cell. The input is defined to be the initial content of the tape. More formally, M is defined as follows: It is a seven tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where Q is the finite set of *states* of the finite control, $\Sigma \subsetneq \Gamma$ is the alphabet of *input symbols*, Γ denotes the set of *tape symbols*, $q_0 \in Q$ is the *initial state* of the finite control, B is the *blank symbol* and $F \subset Q$ denotes the *final* or *accepting states.* The semantics of the *transition relation* δ are as follows: Given a state q and a tape symbol a, the value of $\delta(q, a)$ is a set of triples $\{(p_1, Y_1, D_1), \ldots\}$ with p_i being the next state in Q, $Y_i \in \Gamma$ is the symbol written to the cell being scanned, whereby a is replaced, and D_i is the direction of the movement of head, thus it is either "left" or "right". In case of a deterministic finite control the set only contains a single element, whereby the relation shrinks to a well defined function.

We say that M *accepts* a language L if and only if M *halts*, i. e. there exists some sequence of transitions such that it enters an accepting state after reading w for all $w \in L$. Note, that this sequence is by definition unique in case of deterministic Turing machines, as well as any sequence leading to rejection. On the other hand, non-deterministic Turing machines are assumed to "guess" the correct next step in each transition as there might as well be paths leading to rejection of words in L or infinite runs.

1.1.2 Problems from a formal perspective

In theoretical computer science there is no syntactic difference between a *problem* and a *language* as the former is the question as to whether a word is a member of some language. Thus, when speaking of problems, we mean *decision problems*, as opposed to *functional problems*, where we are not only interested in some yes/no answer but in an effective solution, i. e. an assignment of variables.

Based on this computational model, we can define time- and space complexity classes, i. e. sets of problems that are related in the sense that a deterministic or non-deterministic Turing machine M is able to decide them within a specified time- or space bound. This bound depends on the size of the input. Thus, when speaking of polynomial time, we mean polynomial in the size of the input word. The classes relevant for the further discussion of $\mathcal{AB}$ problems will be explained in the next section.

Before doing so we are to establish *completeness* of some problem L in a class C. This means that L is at least as hard as any other problem in C.

1.1.3 The complexity classes $\mathcal{P}$, $\mathcal{NP}$ and co$\mathcal{NP}$

At first let us consider the class $\mathcal{P}$, i. e. the set of problems solvable in polynomial time by a deterministic Turing machine. Instances thereof are usually considered to be decidable efficiently on actual computing systems [Papadimitriou, 1994]. More formally *solvable* in polynomial time means that any language (a problem) in $\mathcal{P}$ is accepted by some deterministic Turing machine after a number of steps polynomial in the size of the input. On the other hand, solving problems of $\mathcal{NP}$, i. e. the class of problems decidable in polynomial time by *non-deterministic* Turing machines, takes exponential time on any deterministic system, unless a better algorithm is found. This is the famous question of $\mathcal{P} = \mathcal{NP}$, that stands open since Gödel first stated it in his letter to von Neumann.

Boolean satisfiability, which is commonly referred to as SAT, was one of the first problems where $\mathcal{NP}$ completeness was proven [Cook, 1971].

Along with each class one can define its co-class, that is the set of complements of all languages contained in the class. In case of $L \in \mathcal{P}$ this does not change much as the output of some Turing machine deciding L just needs to be inverted, which can be done easily for deterministic Turing machines by swapping the results. However, it is impossible for non-deterministic Turing machines as we only assume them to guess the correct sequence for accepted words, but make no assumption for words to be rejected.

As SAT is complete for $\mathcal{NP}$, so is UNSAT for co$\mathcal{NP}$, that is the question whether there is *no* solution of some Boolean formula. Most probably this is even harder than deciding satisfiability, because we need to prove unsatisfiability for each possible solution.

The relationship of the complexity classes mentioned here is depicted in Figure 1.2. Note, that apart from $\mathcal{P} \subset \mathcal{NP} \cap \mathrm{co}\mathcal{NP}$ none of the relations have been proven and none of them at all has been shown to be strict. For further information on this topic and further references take a look at the Complexity Zoo, `http://qwiki.caltech.edu/wiki/Complexity_Zoo`.

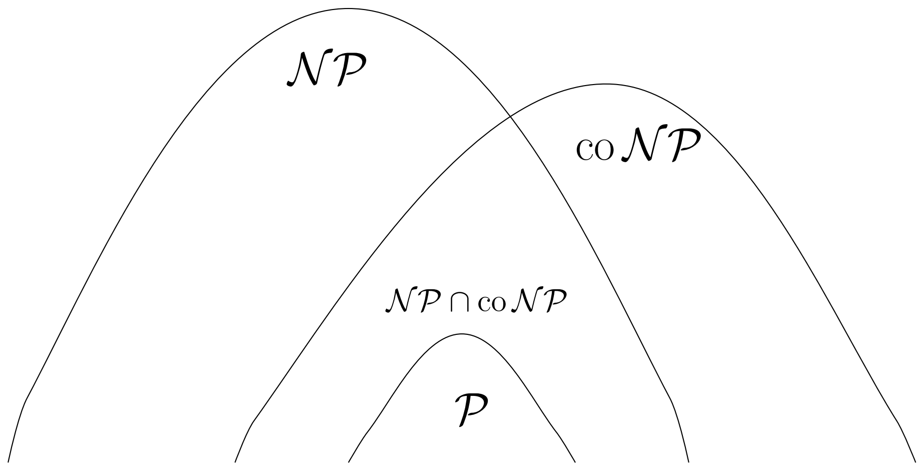

Figure 1.2: A possible relationship of $\mathcal{P}$, $\mathcal{NP}$ and co$\mathcal{NP}$

1.1.4 Oracles

The complexity theoretic thought experiment of *oracles* is meant to provide an algorithm with an "instantaneous correct answer" [Papadimitriou, 1994] to a question of some complexity class, which defines the power of that oracle. Calls to oracles allows us to easily model subroutine calls, however, no knowledge is required on how this subroutine works.

1.2 Linear programming

Let $\mathbb{R}$ be the set of real numbers and let $\mathbb{Z}$ denote the set of integers. The question of linear programming is to find an assignment to $x \in \mathbb{R}^n$, such that the *objective function* $c \cdot x$, for some $c \in \mathbb{R}^n$, is minimal and a set of m constraints is satisfied. The latter are denoted by

$$A \cdot x = 0 \quad A \in \mathbb{R}^{m \times n}.$$

Integer programming deals with the restriction of the entries of A and x to $\mathbb{Z}$. The *standard form* of denoting such problems is as follows:

$$\begin{aligned} &\min cx \\ \text{s.t.}\quad & Ax = 0 \\ & x \geq 0 \end{aligned}$$

This isequivalent to the general form of linear programs (LPs), where any combination of inequalities and equations is allowed. A complete treatment of this matter can be found in, e. g. [Papadimitriou and Steiglitz, 1982].

Whereas the restriction to integers yields an $\mathcal{NP}$ complete problem, the real or rational problem is commonly solved using the Simplex algorithm, despite its worst-case exponential runtime, because it showed to behave well in all common cases and usually converges faster than polynomial time algorithms, such as the ellipsoid method [Khachiyan, 1979].

1.3 Non-linear arithmetic problems

In case of general non-linear optimisation problems the specification is as follows:

$$\begin{array}{rrcl} \min & f(x) & & \\ \text{s.t.} & g^L \leq & g(x) & \leq g^U \\ & x^L \leq & x & \leq x^U \end{array}$$

Thereby $f(x) : \mathbb{R}^n \to \mathbb{R}$ denotes the objective function and $g(x) : \mathbb{R}^n \to \mathbb{R}^m$ are the non-linear constraints, that have upper (g^U) and lower (g^L) bounds.

To clarify this definition, an example shall be given:

$$\min_{x \in \mathbb{R}^4} \; x_1 x_4 (x_1 + x_2 + x_3) + x_3$$
$$\begin{array}{rcl} \text{s.t.} \quad x_1 x_2 x_3 x_4 & \geq & 25 \\ x_1^2 + x_2^2 + x_3^2 + x_4^2 & = & 40 \\ 1 \leq x_1, x_2, x_3, x_4 & \leq & 5 \end{array}$$

An optimal solution would be

$$x = \begin{pmatrix} 1.0 \\ 4.743 \\ 3.821 \\ 1.379 \end{pmatrix},$$

but finding such a solution might depend a lot on a starting point that must be given to all kinds of solvers of this problem. For details on this problem refer to, e. g. [Cooper, 2005].

1.4 On the hardness of $\mathcal{AB}$ problems

The decision problem of linear $\mathcal{AB}$ problems is $\mathcal{NP}$ complete as it can be solved by a Turing machine deciding Boolean satisfiability using an additional $\mathcal{P}$ or $\mathcal{NP}$ oracle for arithmetic constraints over real or, respectively, integer variables. As neither of these oracles pushes the computational power of the model, the problem can as well be decided by some other non-deterministic Turing machine as a whole. Note, that the related functional problem of providing a single solution induces only an at most polynomial overhead and thus a single solution can be provided within the same time bounds.

Non-linear problems, however, cannot be classified as simple as that because, as Abel proved in 1827 [Pan, 1997], there is no closed formula to solve polynomial equations of degree greater than four. Thus all approaches must resort to some numerical algorithms whose runtime depends on the desired accuracy.

1.5 Conjunctive normal form

In contrast to *disjunctive normal form* (DNF), a Boolean formula in *conjunctive normal form* (CNF) is constituted of clauses connected by a logical "AND". Each clause is composed of *literals* joined by a logical "OR". A literal, in turn, is a Boolean variable and may be prefixed by a logical "NOT". A DNF formula, on the other hand, has clauses joined by "OR", composed of "AND"-ed literals. One should note, that the satisfiability of DNF formulae can be computed in linear time, because each clause can be looked at in isolation. However, the conversion of CNF formulae to DNF yields an exponential blow-up and thus is usually infeasible.

Boolean formulae in CNF format are also what SAT solvers usually accept, thus an efficient conversion of arbitrary first-order-logic formulae is desired. For details of this algorithm and possible optimisations refer to, e. g. [Nonnengart and Weidenbach, 2001]. However, an example shall be given. Consider the formula

$$(\neg b_1 \vee b_2) \Rightarrow (b_3 \wedge b_4).$$

The equivalent CNF formula is obtained by first eliminating the implication:

$$\neg(\neg b_1 \vee b_2) \vee (b_3 \wedge b_4)$$

Next, de Morgan's rules are applied and the negations are shifted towards the literals:

$$(b_1 \wedge \neg b_2) \vee (b_3 \wedge b_4)$$

We have now actually obtained a DNF formula, which can be rewritten to a CNF formula using the distributive law:

$$(b_1 \vee b_3) \wedge (b_1 \vee b_4) \wedge (\neg b_2 \vee b_3) \wedge (\neg b_2 \vee b_4)$$

Chapter 2

A framework for $\mathcal{AB}$ problems

To be able to tackle instances of the class $\mathcal{AB}$, as defined in Chapter 1, we developed an extensible Open Source solver system. In this chapter the architecture of our framework is explained and examples of how to use it are given. Furthermore, we describe the external solvers used, and their capabilities.

2.1 Design

Despite the problem we are trying to solve being computationally so hard, efficient solvers for the Boolean and arithmetic domains have been developed to solve most of the effectively occurring instances. Thus our aim was to combine those existing solvers to form a system for $\mathcal{AB}$ problems. To do so, we designed a layered framework, which is sketched in Figure 2.1.

Our system allows for the representation of arbitrary Boolean and arithmetic operations. Thus we could even add arithmetic operators other than $+, -, \cdot$ or $\div$, such as, e. g. sin().

2.1.1 Input layer

The top level of our framework is the interface to the user, which is currently only provided by a parser. It is, however, meant to be left out in case of applications that intend to link the framework as a library.

The parser accepts an extended version of the DIMACS CNF format (see [DIMACS, 1993] for a full specification), whereby we preserve compatibility to existing SAT solvers as we only added semantics to specially crafted comments. On the other hand the user is not forced to learn yet another file syntax. Thus the Boolean part of an $\mathcal{AB}$ problem is taken care of by

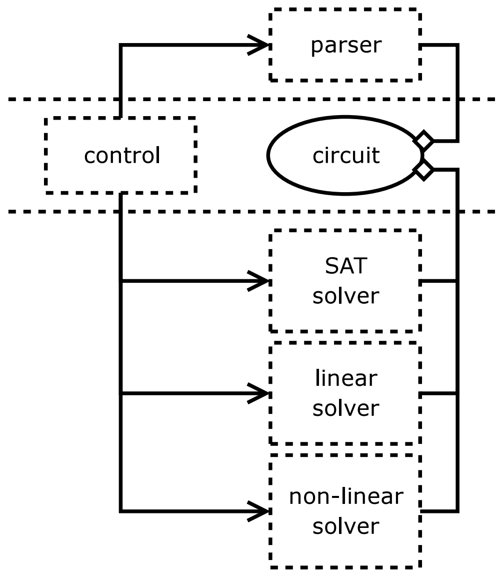

Figure 2.1: Layered approach

DIMACS. Consider, e. g. the formula $(\neg b_1 \vee b_2) \wedge (b_3 \vee \neg b_2)$, which is encoded as shown below:

```
p cnf 3 2
-1 2 0
3 -2 0
```

In DIMACS CNF format commentary lines start with a single “c”, which, in case of our extension, must be followed by the newly defined keyword “def”. To be able to declare integer or floating point expressions, “def” must in turn be followed by “int” or, respectively, “real”. To express the mapping of arithmetic expressions to Boolean variables, as defined in the introduction of $\mathcal{AB}$ in Chapter 1, we resort to the positive integer representing a Boolean variable in DIMACS syntax. As we have thereby established all required definitions, the rest of line is meant to contain the arithmetic comparison, written down using the operators `+, -, *, /, <, <=, =, >=, >`. As a whole, such a line looks like

```
c def real 1 a * x + 3.5 / ( 4 - y ) + 2 * y >= 7.1
```

The sole purpose of the parser is the conversion of textual input to a format acceptable by the core of our framework, where it can be handled efficiently and by means of objects.

2.1.2 The core

We considered the model of integrated circuits as the appropriate real-world equivalent to $\mathcal{AB}$ instances, because all Boolean and arithmetic operators can be seen as gates. This input is either directly connected to one of the input pins or the result of some other gate. The resulting object hierarchy is thus an instance of the Composite Pattern [Gamma et al., 1994].

In Figure 2.2 the model of the above input lines is drawn, which correspond to the following system of mixed arithmetic and Boolean constraints:

$$\begin{aligned} &\left(\neg\left(a \cdot x + \frac{3.5}{4-y} + 2y \geq 7.1\right) \vee b_2\right) \\ \wedge \quad &(b_3 \vee \neg b_2) \end{aligned}$$

The figure shows an artificial operation named “PROP” that represents the mapping of Boolean variables to arithmetic comparisons (see Section 2.4 on page 16 for further details). To avoid confusion with numbers, the Boolean variables $\{1, 2, 3\}$ were renamed to $\{b_1, b_2, b_3\}$, which happens within our application in a similar manner.

The resulting input pins of the circuit correspond to the variables of the $\mathcal{AB}$ problem, whereas the circuit offers only a single output pin representing the truth value of the problem for a given assignment, which may be “true”, “false” or “don’t know” in case some variables have not been assigned to.

It should be noted that a circuit object is self-contained in the sense that all necessary evaluations are done internally. As such, given some circuit, it suffices to pass around sets of variable assignments, apply them to the circuit and request the output value.

For the ease of use we keep track of variable names assigned by the user, whereby we can also provide readable solutions once they are found. Uniqueness of these variables within a circuit is guaranteed by a so called variable pool, which is part of each circuit. Additionally, the implementation of variables within our framework allows for clear distinction between real and integer arithmetic and thus for an efficient implementation of each of them.

2.1.3 Interface to external solvers

A crucial point of our approach is the integration of external solvers, which must be as tight as possible to allow for an efficient implementation, however, it must still be as loose as to make solvers interchangeable. We try to achieve the best possible results using a well defined interface as described in Section 2.3.2 on page 15. Each instance of this interface must itself take

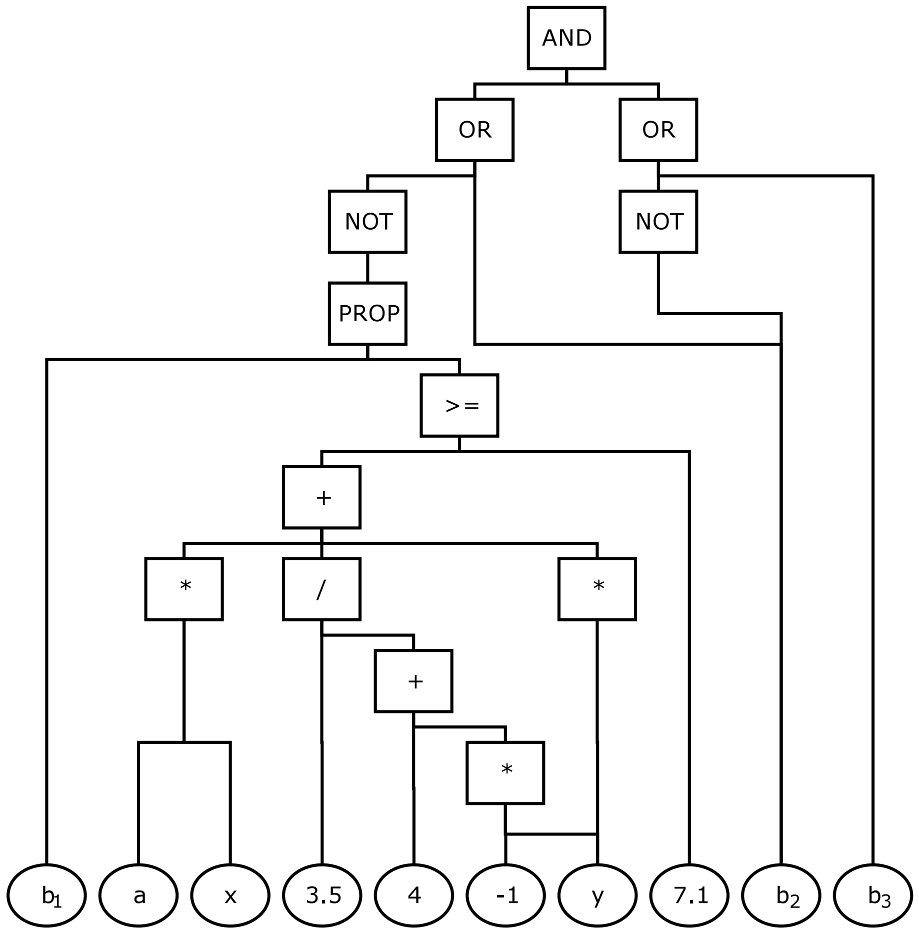

Figure 2.2: The internal representation

care of the efficient application of the underlying solver, but the main task remains the conversion of the circuit to the format the solver requires. To this end we make heavy use of the Visitor Pattern to extract the parts of the circuit relevant for each solver.

2.2 A short glance at existing solvers

In this section we provide a detailed explanation of the external solvers we are using and add some notes, why each of them is particularly useful for our system.

2.2.1 SAT solvers

Current solvers of the Boolean satisfiability problem rely on the DPLL procedure ([M. Davis and H. Putnam, 1960], [Davis et al., 1962]) extended by various optimisations and heuristics. These algorithms enable them to solve even very large instances in reasonable time and the SAT competition (`http://www.satcompetition.org`) pushes further improvement. Whereas nearly all of these solvers accept input in DIMACS syntax, their output varies largely. Thus an important task of each solver's interface is to parse the output provided by the solver and rewrite it to constitute an assignment to the circuit.

Due to the fact, that our algorithm, as described in Section 2.4 on page 16, possibly requires all solution of the Boolean problem we started out using the LSAT [Bauer, 2005] solver, which is capable of computing all solutions in a single run.

Next, support for GRASP [Marques-Silva and Sakallah, 1996] has been added, which is faster that LSAT for some problems, but proved not to be as stable and aborted on problems with a high number of solutions. Furthermore, GRASP prints solutions in a compressed format, i. e. for each set of solutions where only some variables are fixed and the others may be "true" and "false", only the fixed ones are printed. Even though improvements are being considered in this area, the current implementation requires the full set of solutions and thus the interface needs to build the remaining parts.

One of the best known solvers is zChaff [Moskewicz et al., 2001], which can cope easily cope with hard instances, but for our application it has the disadvantage that it does not offer any means to obtain all possible solutions at once. Consequently it has to be called again and again while adding the negated result as a new clause until the system becomes unsatisfiable. This loop incurs a notable overhead of invoking the process in each iteration, thus this solver should only be used for very hard instances. However, there is also a C++ library delivered with the solver, whereby the forking overhead is eliminated and our system can fully benefit from zChaff's strength.

As the computation, and even more so the storage, of all solutions of the Boolean system at once is likely to be infeasible on actual systems, we added code to return only a single solution at each call to the interface to the zChaff library – the same feature will be added to other solvers in the future.

2.2.2 Linear solvers

Linear optimisation problems play an important role in business and thus there is a whole load of linear solvers, yet very few of them are freely available. Whereas `lp_solve` [Berkelaar et al., 2005] is fairly well known in the area of free software, the Common Optimization Interface (COIN) [Lougee-Heimer, 2003] seems to be a lot more professional and includes integer solvers that perform very well. As we already had some experience using it, embedding it in our framework was a straight forward task. Furthermore COIN itself provides interfaces to other, mostly commercial, linear solvers that could be used via a single interface.

2.2.3 Non-linear solvers

As opposed to linear programming, code for non-linear programming is still very seldom and mostly restricted to Fortran functions. One of the freely available non-linear solvers offering a C++ interface is the Interior Point Optimizer (IPOPT) [Wächter and Biegler, 2006], but, as do most of the solvers in this area, it requires a lot more information about the problem, including derivatives, which we compute using CppAD [Bell, 2003], a tool for automatic differentiation.

2.3 Aspects of the implementation

The whole project contains more than 12,000 lines of code, half of which accounts for the core. The build environment is managed using GNU autotools [Vaughan and Tromey, 2000], which supported a structured organisation of the source files. Thus, even though urrently the control task is completely implemented in the `main` function, the movement of which to other parts of the code soon to facilitate the use of the system as a library will be easy.

2.3.1 Reasons for using C++

The framework has been implemented using C++ as it offers, e. g. templates, which are handy when it comes to implementing the operations like addition or multiplication for different data types, which also holds for the implementation of variables. Furthermore it features many ways of optimisation [Meyers, 1998], which we made heavy use of as far as the core is concerned, whereas the solver interfaces still demand a lot of profiling and improvement. As our system continues to be a growing software project, we also benefit from the scalability of the language [Lakos, 1996].

2.3.2 Solver interface

To get a better idea of the algorithm we present in the next section, we briefly discuss the interface to external solvers in terms of the methods. Note, that we omitted any implementation details, such as constness or namespaces.

```
unsigned solve( CircuitConcept *, SolutionSet &,
                SolutionSet & );
```

This function is the only one required for communication between the algorithmic control and the solver itself. Whereas the `CircuitConcept`, which is the representation as discussed in Section 2.1.2 on page 11, is an input only, the `SolutionSet`s are also ways to provide output in the sense that each solver it meant to store, if applicable, a feasible solution and computed conflicts in those objects. The return value provides some status information, if supported by the solver interface. To obtain information on whether something is supported or not, the function

```
unsigned capabilities();
```

is part of the interface. Besides the definition of applicable status codes, the returned value includes information on the arithmetic or Boolean domains supported by the underlying solver.

One of the major design goals were means of providing access to solver-specific heuristics. As such, each solver may be configured on the command line without any need to modify the `main` procedure for each option. To this end, each solver must have a unique name, which can internally be obtained using the

```
string name();
```

function. For the actual parameters, the following methods are provided:

```
  void set_option( string & name, Parameter const & p );
  string list_options();
```

Both of which are meant to be used by any kind of front-end, which can thereby list the available parameters and set any of them.

2.4 The basic algorithm

The current control loop, which is run until either the desired number of solutions has been found or no further valid assignments can be obtained, is very simple and improvements as explained in, e. g. [Hofstedt, 2001] are to be considered.

The basic idea is as follows: The SAT solver is queried for all solutions to the Boolean part of the problem and possibly returns a set of valid assignments. Each of them is then applied to the circuit, whereby all Boolean variables attain values, as well as the propositional gates, which take care of the mapping to arithmetic comparisons. They do so by ensuring the equality of the value of the Boolean variable, which is attached to the first input, and the truth value of all attached arithmetic comparisons. The gate evaluates to "true" if and only if the value of the Boolean variable is "true", as well as the truth value of all attached comparisons.

Using the preconditions provided by the current assignment, the arithmetic solvers are called, whose input is built using the arithmetic terms and a comparison operator depending on the value supposed by the propositional gate and the operator of the original problem. To clarify this idea, reconsider the the Boolean variable b_1 in Figure 2.2 on page 12 and the related comparison $\geq$. Under the assumption that b_1 were assigned "false", this would result in the inversion of $\geq$ to $<$.

Let us elaborate on this workflow using the following example of mixed Boolean, linear and non-linear constraints. The set of assignments found is directly taken from the output provided by our system – the complete session is printed in Appendix B. The $\mathcal{AB}$ instance reads as follows:

$$\begin{array}{ll} & ((i \geq 0) \wedge (j \geq 0)) \\ \wedge & (\neg(2i + j < 10) \vee (i + j < 5)) \\ \wedge & \left(a \cdot x + \dfrac{3.5}{4 - y} + 2y \geq 7.1\right) \end{array}$$

Spelled out in terms of our input format, the following input must be provided to our solver:

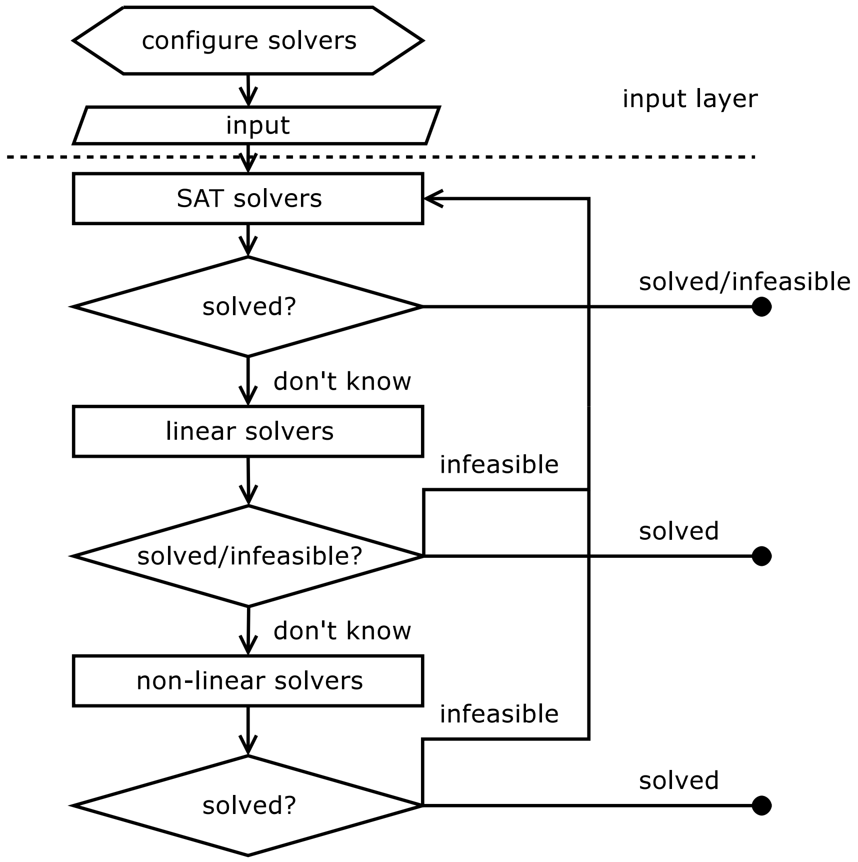

Figure 2.3: Control loop

```
p cnf 4 3
1 0
-2 3 0
4 0

c def int 1 i >= 0
c def int 1 j >= 0
c def int 2 2*i + j < 10
c def int 3 i + j < 5
c def real 4 a * x + 3.5 / ( 4 - y ) + 2 * y >= 7.1
```

The solver is then called as, e. g.

```
ABsolver -f inputfile -ssat:zchafflib -Sall_solutions=false \
  -sl:coin -Sverbosity=0 -snl:ipopt
```

This set of parameters tells our tool to read the input from `inputfile` and to use the solvers `sat:zchafflib`, which is the library version of zChaff, and `l:coin`, which is the COIN linear solver, and the non-linear solver IPOPT, internally named `nl:ipopt`. Furthermore the solver-specific options `verbosity` and `all_solutions` are set. For a detailed description of these and all further options run `ABsolver --help`.

Back at the workflow, the next step is processing the input, whereby the circuit is built by the parser. This circuit is then read by the interface of the SAT solver and, in this case, converted to a set of instructions for the zChaff library. The first solution returned by the SAT solver is

$$b_1 = true, b_2 = false, b_3 = true, b_4 = true.$$

As the truth value of the complete system is still "don't know", the linear solver is called to solve the system

$$\begin{aligned} i &\geq 0 \\ j &\geq 0 \\ 2i + j &\geq 10 \\ i + j &< 5 \end{aligned}$$

This linear system, however, is infeasible and thus the control gets back at the SAT solver, which is asked for another solution. It provides

$$b_1 = true, b_2 = false, b_3 = false, b_4 = true,$$

which implies

$$\begin{aligned} i &\geq 0 \\ j &\geq 0 \\ 2i + j &\geq 10 \\ i + j &\geq 5 \end{aligned}$$

with the solution of $i = 5, j = 0$. Yet, the overall value is still "don't know" as the feasibility of the non-linear inequality has not been proven. This is, however, an easy task for IPOPT, which provides $a = 0, x = 0, y = 2.52904$ as a valid assignment, whereby the mixed system attains "true".

2.5 Extensions and optimisations

Lists of solvers. Despite the strength of existing solvers in the various domains one should be aware of the fact that not all of them are equally well suited for all instances and might even abort on hard problems. Thus we added support for the use of a list of solvers at each stage, which are queried until one of them provide a decent result, i. e. either is able to tell infeasibility or to return a solution. To this end we add that infeasibility is not defined as returning no solution, but rather via the status code provided by the `solve` method. This is due to the fact that not all arithmetic solvers are able to reliably prove unsatisfiability, but might instead stop after a certain number of iterative steps.

Conflict learning. On the other hand, if an arithmetic solver is able to prove infeasibility of the current instance passed to the solver, it can be extended to compute a smaller unsatisfiable subset of the arithmetic expressions. This is done by sequentially removing constraints until a satisfiable system is obtained, thus the last expression that has been removed must be part of a conflict. This process is continued until removal of all constraints has been attempted once, so this takes as many steps as the number of constraints.

2.6 Domain-specific issues

2.6.1 Exponential number of Boolean assignments

In case the Boolean part of the $\mathcal{AB}$ instance at hand has a huge number of solutions, having the SAT solver find all solutions at once will likely result in enormous memory consumption and is thus infeasible. To circumvent this issue, the interface to the zChaff SAT library has been extended to compute only a single assignment each time its `solve` method is called. Still, the result is retained to guarantee that the is solution is only returned once. A similar extension is planned for all other SAT solvers as well.

2.6.2 Defining an objective function

Even though we are not interested in finding an – in any way – optimal solution to the arithmetic problem, the external arithmetic solvers require use to specify an objective function of the optimisation problem, as defined in Section 1.2 on page 5. In case of the linear solver we resort to the constant zero function, which causes the Simplex algorithm to stop as soon as a feasible point has been computed, because no improvement of the object value is seen in any search direction.

For the non-linear solver, on the other hand, we refrained from doing so as experiments proved a very bad performance. There we resort to the absolute value of the constraint violation and, in case of IPOPT, we are also required to provide derivative information and can use it as a further measure to guide the solver to a feasible solution.

2.6.3 Starting point for non-linear optimisation

As noted in Section 1.3 on page 6, the non-linear solvers require us to provide a starting point. The current implementation tries to provide 0 as an initial value for all variables, but searches for alternatives by adding 0.1 to each of the variables, if the prior assignments would have resulted in divisions by zero.

2.6.4 Exponential number of arithmetic systems

Given a proposed truth value of some arithmetic comparison, we possibly need to invert the comparison operator, as discussed in Section 2.4 on page 16. Consequently, in case of equalities, we obtain expressions that must *not* be equal. Inevitably this results in an exponential number of arithmetic *systems* as these hyperplanes split the n-dimensional space into an exponential number of simplexes and feasible solutions could be located in any of these. Even though the hyperplanes form a null set for real arithmetic (see any textbook on analysis and linear algebra, such as [Fischer, 2002], for an exhaustive treatment of these topics), this is not true for integer and finite precision computation and the methods suggested by [Bozzano et al., 2005] are, in our opinion, not fully applicable. However, due to the way we defined the objective function for non-linear arithmetic in the IPOPT solver interface, the idea of [Bozzano et al., 2005], that is simply removing any "not-equal" constraints and only verifying that possible solutions do not violate the constraints, might be viable.

Additionally it should be noted that the conflict deduction is disabled in such cases as only those conflicts would be valid that can be found in all of the systems.

2.6.5 Precision in floating point comparisons

Due to the limited precision of data types for real arithmetic, such as `double`, testing for equality might yield unexpected results. For a detailed treatment refer to, e. g. [Kahan, 1997], but as a simple example just consider that $1.6 \cdot 3$ and 4.8 are off by approximately 10^{-16}. Thus such comparisons must be implemented as tests for the absolute difference being below some threshold.

For the same reasons working with strict inequalities on computer systems is different than in theory, because for an expression like $x > 1$ there might be a solution that is slightly greater than 1 with the difference being beyond the precision of the data type. Due to the fact that the used external routines only deal with non-strict inequalities we need to convert strict inequalities by adding or, respectively, subtracting some threshold which depends on precision of the solver, as it should be as low as possible. The resulting conversions are, e. g. as follows:

$$\begin{array}{rlcl} \text{Integer:} & x + y < 10 & \rightarrow & x + y \leq 9 \\ \text{Real:} & a + b > 1.5 & \rightarrow & a + b \geq 1.500001 \end{array}$$

Besides the dependence on the solver's precision, the desired threshold might be problem specific and can be specified as an option to the arithmetic solvers.

Chapter 3

Results

In this chapter we present the main experimental results of our work, as well as a set of applications.

3.1 Use cases

In the following pages we list a few use cases, where aspects of our implementation allow for either totally new applications or performance improvements over existing solvers. Even though our system has not been planned with those particular applications in mind, the design showed to allow for unique features that make the tool applicable in various areas. One should consider that the domain of verification is what satisfiability modulo theories aims at, however, the problem as described in Section 3.1.4 on the following page was not solvable by any of the competitors. Still, we must be aware of the fact, that tools like MathSAT or CVC Lite [Barrett and Berezin, 2004] perform very well when problems are restricted to linear real arithmetic. Furthermore, CVC Lite is even able to deal with polynomial expressions and thus covers parts of the non-linear domain.

3.1.1 Diagnosis

The task of diagnosis is, given an observation that deviates from the expected outcome, to find faulty components that explain such a behaviour. The related background and methods to perform this task were introduced by Reiter [Reiter, 1987], another approach using propositional logic was presented in [Bauer, 2005]. In either case, the correctness of numerical values is tested by monitors, as the Boolean reasoning can only take care of truth values.

Using mixed logical and arithmetic solvers instead allows for modelling the diagnostic task directly, but the tools must meet a few requirements: They must be capable of printing a solution as satisfiability is already known, given the observation of an error. Furthermore all solutions must be provided to facilitate the computation of minimal conflicts. Both of these requirements are met by our tool.

3.1.2 Test case generation

Testing software and hardware systems is commonly performed using a set of input vectors and checking correctness of the corresponding output. To obtain reliable results that offer full statement coverage or even full path coverage the effective system has to be analysed and modelled. Several approaches are discussed in [Broy et al., 2005], which also offers a more extensive reference on this topic.

The applicability of mixed Boolean and arithmetic solvers has already been shown in [Fallah et al., 2001], where these tools were proposed to support directed test case generation for a selected path. As we are able to compute all solutions of a mixed system, we can examine all paths at once and return the set of test vectors required for full path coverage.

3.1.3 Puzzles and games

Even though all interesting games are at least $\mathcal{NP}$ complete and can thus be encoded as Boolean satisfiability problems, this approach may be quite difficult, especially if numbers other than 0 and 1 are involved. Instead it is a lot easier to rewrite the problem as a mixed arithmetic and Boolean problem.

As an example, consider the following problem of a coloured cube, given nine red, nine blue and nine white pieces: Form a $3 \times 3 \times 3$ cube, such that every diagonal, every space diagonal and all vertical or horizontal rows contain exactly two different colours.

As a possible encoding of this problem collapses to a system of linear integer constraints, we benefit from the strength of our linear solvers, as seen in Section 3.2 on page 27.

On the other hand, for puzzle developers it is interesting to verify, whether a problem has only a single solution or more of them. This is supported by our approach by simply asking the solver to find two solutions.

3.1.4 Verification

Figure 3.2 depicts an aspect of the the steering control system of a car, including some black boxes whose function is known and includes Boolean and arithmetic operations. These are sketched as follows: The controller obtains floating point data from a yaw sensor, a lateral acceleration sensor, speed sensors at each wheel and the steering angle. Using these inputs, it is due to the system to detect over- or under-steering and to direct a step motor accordingly to keep the car stable, whereby critical driving situations should be avoided.

The task is to prove that there are no faults in this design that could lead to hazardous driving situations. To do so, the controller and the environment, i. e. the car, need to be modelled. Whereas the model of the controller can be linearised with some effort, the model of the environment inevitably yields non-linear expressions. As such the industry relies on extensive simulation to ensure the correctness of the controller, because tools like SCADE cannot cope with non-linear problems.

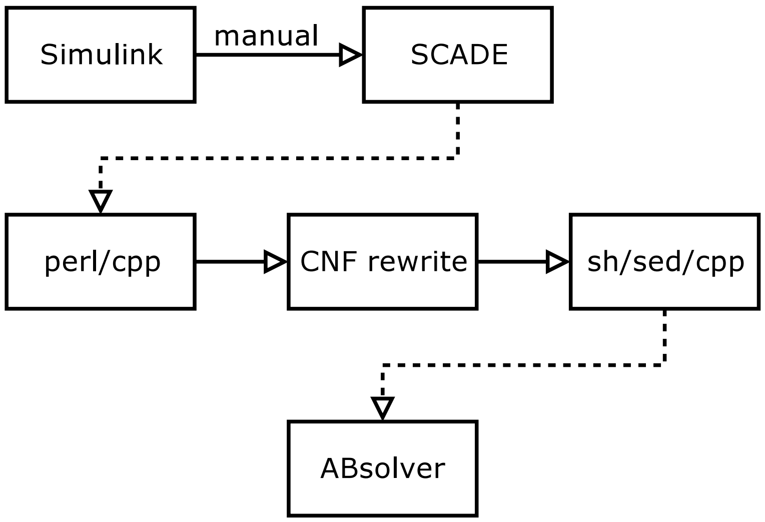

Figure 3.1: Workflow of the conversion

The extensible design of our tool and the resulting integration of non-linear solvers allowed the expression of the complete system, yet a conversion from Mathlab Simulink was required. The procedure is depicted in Figure 3.1 and involved 21 stages, but was done automatically apart from the import into SCADE. The result were 976 Boolean CNF clauses and 24 non-linear expressions that were obtained within 3 hours on a 3.2 GHz Intel Xeon system.

Figure 3.2: Aspect of a controller of a car steering

The detailed workflow is as follows: As SCADE writes each component of the controller into a file, these are combined using the C preprocessor and some dependency deduction. In the next steps, expressions such as "if–then–else", "Abs" and "<>" are replaced to obtain valid $\mathcal{AB}$ expressions. After splitting the Boolean and the arithmetic parts, the logical formulae must be rewritten to CNF, which causes a huge blow-up. As the size of intermediate expressions exceeded 1 GByte, we were to write our own tool, which writes the formulae to the hard drive and only works on subexpressions, whereby we are able to work at a constant memory consumption of approximately 25 MBytes. Before generating the final output, the Boolean formulae are checked for redundancies and tautologies, which allowed for the reduction to a size of 20 KBytes.

3.2 Benchmarks

Table 3.1 summarises the times it took us to solve some problems. A detailed description of each benchmark is given below. The second and the third benchmark were run on a x86 system with 2.1 GHz and 1024 MBytes of RAM, because MathSAT is only distributed in binary form. The other experiments were conducted on a 1.5 GHz PowerPC-based system with 1280 MBytes of memory. In all cases we used the command line parameters given in Section 2.4 on page 16.

Name	Time	Competitor	Time
Presentation	0.2 s	-	-
Cube	12 s	MathSAT	304 m
MathSAT/DTP_k2_n35_c175_s1	410 m	MathSAT	0.1 s
Steering control	59 s	-	-
ISCAS/s344	0.2 s	zChaff	0.03 s

Table 3.1: Benchmark summary

Presentation. This is the example described in Section 2.4 on page 16. As it contains a non-linear part, we had no competitors to test.

Cube. An encoding of the puzzle explained in Section 3.1.3 on page 24, the resulting system had 634 linear integer inequalities over 81 variables. In contrast to out competitor, MathSAT, we benefit from the powerful linear integer solver provided by COIN.

MathSAT/DTP_k2_n35_c175_s1. This is an artificial problem, taken from the MathSAT benchmark suite, which should soon be available again at `http://mathsat.itc.it/benchmarks.html`. This particular instance has 35 real valued variables in 350 inequalities. These are part of 175 Boolean CNF clauses with two literals each. The Boolean system on its own is trivial and has 3^175 possible solutions. Without a conflict strategy, finding a solution is merely infeasible.

Steering control. This is the industrial application as discussed in Section 3.1.4 on page 24.

ISCAS/s344. This example has been taken from the ISCAS89 benchmark suite, which models real logic circuits and their faults, and is available from `http://www.visc.vt.edu/~mhsiao/ISCAS89/s344.bench`. It is merely used to measure the overhead of our parser, which accounts for most of the 0.2 seconds as computing two solutions takes only 0.03 seconds longer.

Chapter 4

Conclusions

In this thesis we presented a new approach towards the problem of mixed arithmetic and Boolean constraint systems. The importance of this kind of problems, which we defined to be the class of $\mathcal{AB}$ was underlined by the set of use cases explained in Chapter 3 on page 23. However, this list is most probably still far from complete.

To be able to tackle the new set of problems we developed an extensible framework that offers new features, such as the applicability of domain specific heuristics or computing any number of solutions of the system, which my even come in handy for Boolean-only problems.

The benchmark results (cf. Section 3.2 on page 27) are promising, however, a lot more testing is required which was impossible due to the time constraints of this thesis as it requires a lot of format conversion. This will be a lot easier once the new parser has been implemented (see Section 4.2 on the next page).

4.1 Contribution

Even though the primary goal of this thesis was a solver for mixed logical and linear systems, the idea of doing non-linear arithmetic came up very early. This was due to the real-life problem of the car steering control system, which could not be solved by any of our competitors. The extensible design proved very useful as there was no need for extensions within the core of our system to support these constraints.

It was again the extensibility that was useful as the hardness of some benchmarks required the addition of zChaff as a more powerful SAT solver, which was very little work.

To a very different end, the system is intended for the working programmer and all interfaces are well documented, which will be helpful for further extensions that are not yet predictable, but will surely be required. Although the area of satisfiability modulo theories and the corresponding tools seem to provide a lot of functionality, our tool will be used by at least two other diploma thesis in the next months. Therefore we conclude that the existing tools do not satisfy the needs of the potential users.

4.2 Possible extensions

As the framework is published under an Open Source license and available at `http://absolver.sf.net`, all programmers are welcome to contribute new ideas. However, our own list of open tasks includes many desired improvements, a few of which shall be presented here.

One of most important tasks is the implementation of a parser for the SMT format [Ranise and Tinelli, 2005], whereby we will be able to do a lot more testing without the need to convert each of the benchmarks to our input format. This will gain even more importance when it comes to testing new conflict strategies. To this end, the ideas of [de Moura and Ruess, 2002] are to be studied and implemented. Furthermore, there is basically no need to have a complete assignment of the Boolean problem, if some subset already guarantees that the result will be "true", which is perfectly supported by GRASP. This would lower the number of Boolean solutions and reduce the number of arithmetic constraints that need to be satisfied.

In the area of arithmetic solvers a lot more research and testing of external solvers is required, especially the non-linear part would benefit from that. Even more so, an approach of using equation solvers instead of optimisation software by an introduction of slack variables might provide a lot more efficiency.

Appendix A

EBNF description of the input format

The following grammar is a complete EBNF description of our input format. For a short introduction to EBNF see, e. g. [Marcotty and Ledgard, 1986].

```
unsigned ::= [0-9][0-9]*
integer ::= <unsigned>
           | -<unsigned>
real ::= <integer>
        | <integer>.<unsigned>
edimacs ::= <lines> EOF
lines ::= EOL
          | <lines> <dimacs> EOL
          | <lines> <extension> EOL
dimacs ::= <problem_description>
           | <comment>
           | <clause> 0
problem_description ::= p cnf <unsigned> <unsigned>
comment ::= c <printable characters>
clause ::= <integer>
           | <clause> <integer>
extension ::= c def <integer_decl> <unsigned> <int_eq>
              | c def <real_decl> <unsigned> <real_eq>
integer_decl ::= int
                 | i
real_decl ::= real
              | r
int_eq ::= <int_expr> <comparison> <integer>
real_eq ::= <real_expr> <comparison> <real>
comparison ::= <
```

```
               | >
               | =
               | <=
               | >=
int_expr ::= <integer>
             | <variable>
             | <int_expr> + <int_expr>
             | <int_expr> - <int_expr>
             | <int_expr> * <int_expr>
             | <int_expr> / <int_expr>
             | ( <int_expr> )
             | -<int_expr>
real_expr ::= <real>
              | <variable>
              | <real_expr> + <real_expr>
              | <real_expr> - <real_expr>
              | <real_expr> * <real_expr>
              | <real_expr> / <real_expr>
              | ( <real_expr> )
              | -<real_expr>
variable ::= [a-zA-Z][a-zA-Z0-9_]*
```

Appendix B

Example session

The following listing is the output provided by our solver for the problem described in Section 2.4 on page 16. To gain some more information, the parameter for increasing verbosity (`-V`) has been added twice.

```
$ src/ABsolver -f examples/presentation -VV -ssat:zchafflib \
  -Sall_solutions=false -sl:coin -Sverbosity=0 -snl:ipopt
Trying SAT solver...
done.
Applying 1_bool=1 2_bool=0 3_bool=1 4_bool=1
Value is now DONT_KNOW
```

At that stage the first Boolean assignment has been computed, which is

$$b_1 = true, b_2 = false, b_3 = true, b_4 = true.$$

Yet, the overall truth value is unknown.

```
Trying linear solver... Coin0510I Presolve is modifying 2 integer
  bounds and re-presolving
Coin0506I Presolve 2 (-2) rows, 2 (0) columns and 4 (-2) elements
Cgl0000I Cut generators found to be infeasible!
done.
Linear system is infeasible.
Trying SAT solver... Adding conflict 1_bool=1 2_bool=0 3_bool=1
```

The linear solver was able to prove that the resulting system is infeasible. Thus a conflict is computed and used by the SAT solver.

```
done.
Applying 1_bool=1 2_bool=0 3_bool=0 4_bool=1
```

```
Value is now DONT_KNOW
Trying linear solver... Coin0506I Presolve 0 (-4) rows, 0 (-2)
  columns and 0 (-6) elements
Cgl0004I processed model has 4 rows, 2 columns (2 integer) and
  6 elements
done.
Applying i=5 j=0
Value is now DONT_KNOW
```

A feasible solution of $i = 5, j = 0$ has been found for the linear system, however, the feasibility of the non-linear system is unknown.

```
Trying nonlinear solver... Problem solved
done.
Applying a=0 x=0 y=2.52904
Value is now 1
Satisfying assignment: 1_bool=1 2_bool=0 3_bool=0 4_bool=1 /
  i=5 j=0 / a=0 x=0 y=2.52904
```

The non-linear system was found to be feasible and a valid solution of $a = 0, x = 0, y = 2.52904$ has been returned. As the overall value is now 1, which is equal to "true", the solver finishes and prints the complete solution.

Bibliography

[Barrett and Berezin, 2004] Barrett, C. and Berezin, S. (2004). CVC Lite: A new implementation of the cooperating validity checker. In Alur, R. and Peled, D. A., editors, *Proceedings of the* 16*th* *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag. Boston, Massachusetts.

[Bauer, 2005] Bauer, A. (2005). Simplifying diagnosis using LSAT: A Propositional Approach to Reasoning from First Principles. volume 3524, Prague, Czech Republic. Springer-Verlag.

[Bell, 2003] Bell, B. M. (2003). CppAD: A Package for C++ Algorithmic Differentiation.

[Berkelaar et al., 2005] Berkelaar, M., Eikland, K., and Notebaert, P. (2005). lp solve: Open source (mixed-integer) linear programming system.

[Bozzano et al., 2005] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., and Sebastiani, R. (2005). An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Proceedings of TACAS'05*. Springer.

[Broy et al., 2005] Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., and Pretschner, A., editors (2005). *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer.

[Clarke et al., 1999] Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press, Cambridge, Massachusetts.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*, pages 151–158.

[Cooper, 2005] Cooper, J. (2005). *Working Analyis.* Elsevier/Academic Press, London.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5:394–397.

[de Moura and Ruess, 2002] de Moura, L. and Ruess, H. (2002). Lemmas on demand for satisfiability solvers.

[DIMACS, 1993] DIMACS (1993). Satisfiability: Suggested Format. Technical report.

[Fallah et al., 2001] Fallah, F., Devadas, S., and Keutzer, K. (2001). Functional vector generation for HDL models using linear programming and boolean satisfiability. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 20:994–1002.

[Fischer, 2002] Fischer, G. (2002). *Lineare Algebra.* Vieweg, 13^{th} edition.

[Fortnow and Homer, 2002] Fortnow, L. and Homer, S. (2002). A Short History of Computational Complexity.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts.

[Hartmanis and Stearns, 1965] Hartmanis, J. and Stearns, R. E. (1965). On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306.

[Hofstedt, 2001] Hofstedt, P. (2001). *Cooperation and Coordination of Constraint Solvers.* PhD thesis, Dresden University of Technology.

[J. E. Hopcroft, 2001] J. E. Hopcroft, R. Motwani, J. D. U. (2001). *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, New York, second edition.

[Kahan, 1997] Kahan, W. (1997). Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic.

[Khachiyan, 1979] Khachiyan, L. G. (1979). A polynomial algorithm in linear programming. *Doklady Akedamii Nauk SSSR*, 244:1093–1096.

[Lakos, 1996] Lakos, J. (1996). *Large Scale C++ Software Design.* Addison-Wesley.

[Lougee-Heimer, 2003] Lougee-Heimer, R. (2003). The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community. *IBM J. Res. Dev.*, 47(1):57–66.

[M. Davis and H. Putnam, 1960] M. Davis and H. Putnam (1960). A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215.

[Marcotty and Ledgard, 1986] Marcotty, M. and Ledgard, H. (1986). *The World of Programming Languages.* Springer.

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

[Meyers, 1998] Meyers, S. (1998). *Effective C++.* Addison-Wesley, second edition.

[Moskewicz et al., 2001] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01).*

[Nonnengart and Weidenbach, 2001] Nonnengart, A. and Weidenbach, C. (2001). Computing Small Clause Normal Forms. In *Handbook of Automated Reasoning*, pages 335–367.

[Pan, 1997] Pan, V. (1997). Solving a polynomial equation: Some history and recent progress. *SIAM Review*, 39:187–220.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity.* Addison-Wesley, New York.

[Papadimitriou and Steiglitz, 1982] Papadimitriou, C. H. and Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ.

[Ranise and Tinelli, 2005] Ranise, S. and Tinelli, C. (2005). The SMT-LIB Standard: Version 1.1. Technical report.

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

[Vaughan and Tromey, 2000] Vaughan, G. V. and Tromey, T. (2000). *GNU Autoconf, Automake and Libtool*. New Riders Publishing, Thousand Oaks, CA, USA.

[Wächter and Biegler, 2006] Wächter, A. and Biegler, L. T. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57.

[Wolfman and Weld, 1999] Wolfman, S. A. and Weld, D. S. (1999). The LPSAT Engine and its Application to Resource Planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 310–316.