Christian Herde

# Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure

## Proof Engines for the Analysis of Hybrid Discrete-Continuous Systems

VIEWEG+
TEUBNER

Christian Herde

# Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure

VIEWEG+TEUBNER RESEARCH

Christian Herde

# Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure

## Proof Engines for the Analysis of Hybrid Discrete-Continuous Systems

With a foreword by Prof. Dr. Martin Fränzle

VIEWEG+TEUBNER RESEARCH

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Dissertation Universität Oldenburg, 2010

1st Edition 2011



Editorial Office: Ute Wrasmann | Anita Wilke

Vieweg+Teubner Verlag is a brand of Springer Fachmedien.
Springer Fachmedien is part of Springer Science+Business Media.
www.viewegteubner.de




Cover design: KünkelLopka Medienentwicklung, Heidelberg
Printed on acid-free paper
Printed in Germany

ISBN 978-3-8348-1494-4

# Foreword

Over the past decades, we have seen a slow, but steady popularization of automatic verification technology — once the Holy Grail of formal methods in computer science — that is indicative of the maturity the field has achieved. Only few universities do still hesitate to expose undergraduates to the field's basic methods, the pertinent tools, and their underlying algorithms, and industrial takeover is undeniably gaining impetus, following education with the natural phase delay. Over the years, some basic techniques have become cornerstones of the field, forming identifiable and reappearing building blocks of the plethora of tools that have been built. Among these fundamentals are abstract interpretation, binary decision diagrams, and satisfiability solving, to name just a few. Within this book, you will find an informed account of a number of substantial contributions to the latter field, in particular addressing the domain of satisfiability modulo theories, which has become instrumental to various mechanic verification schemes in hardware and software validation. Despite being based on a PhD thesis, which I had the pleasure to advise, the book elaborates in equal detail on the underlying "folklore" ideas and techniques and on the author's own contributions, complementing both by extensive pointers to open problems and ideas for further research. I hope you as a reader will find it helpful, no matter whether you are a novice trying to understand satisfiability solving for complex-structured arithmetic constraints or are an expert looking for a clear-cut delineation of the techniques developed in the *Transregional Collaborative Research Center AVACS* (Automatic Verification and Analysis of Complex Systems, funded by the Deutsche Forschungsgemeinschaft) from competing approaches.

*Martin Fränzle*

# Acknowledgements

> *In a sense, nature has been continually computing the 'next state' of the universe for billions of years; all we have to do – and, actually, all we* can *do – is 'hitch a ride' on this huge ongoing computation, and try to discover which parts of it happen to go near to where we want.*
>
> *Tommaso Toffoli, 1982*

This thesis marks a major waypoint on my excursion into the field of formal verification and constraint solving, a pleasurable ride along a branch of the 'huge ongoing computation' which turned out to be one of the more exciting ones. I would like to thank all those who made this journey possible:

First and foremost, I would like to thank my supervisor Martin Fränzle. A big thank you, Martin, for all I have learned from you, for your perpetual willingness to discuss any questions and ideas I have had, for your valuable advice, and for giving me the time I needed to finish this work.

Particular thanks go to Bernd Becker for kindly accepting to co-examine this thesis, and to him and the members of his research group for their hospitality and support during my visits to Freiburg. I also extend my thanks to my other committee members Sibylle Fröschle and Andreas Winter for taking time to participate in my defense. Moreover, I would like to express my gratitude to Werner Damm for sparking my interest in safety-critical systems and for offering me my first job in formal verification during an oral exam many years ago.

I am particularly indebted to my friends and colleagues from the *University of Oldenburg*, from the *Transregional Collaborative Research Center AVACS*, and from the *OFFIS Institute for Information Technology* for providing an inspiring and pleasant working environment. I am especially grateful to Andreas Eggers and Tino Teige for outstanding teamwork, for many discussions on research and life, and for the great time we spent together.

Lastly, and most importantly, I am deeply thankful to my grandparents Anna and Friedrich Bödeker who raised me and gave me loving care and support in difficult times. To their memory, I dedicate this thesis.

*Christian Herde*

# Abstract

Due to the increasing use of more and more complex computerized systems in safety-critical applications, formal verification of such systems is of growing importance. Among the most successful methods in formal verification of finite-state systems is bounded model checking (BMC), a technique for checking whether an unsafe system state is reachable within a fixed number of steps. BMC belongs to a class of verification algorithms having in common that the verification task is reduced to the problem of checking the satisfiability of a propositional formula or a series thereof. Though originally formulated for discrete transition systems only, BMC is in principle also applicable to hybrid discrete-continuous systems, which naturally arise e.g. in the field of embedded systems where digital (discrete) controllers are coupled with analog (continuous) physical plants. The BMC formulae arising from such systems are, however, no longer purely propositional, but usually comprise complex Boolean combinations of arithmetic constraints over real-valued variables, thus entailing the need for new decision procedures to solve them.

This thesis deals with the development of such procedures. A key component of the algorithms we present is the DPLL procedure for solving Boolean formulae which are in conjunctive normal form (CNF). As a first contribution, we demonstrate that the acceleration techniques, which enabled the enormous performance gains of Boolean SAT solvers in the recent past, generalize smoothly to DPLL-based procedures for solving systems of pseudo-Boolean constraints, a much more concise representation of Boolean functions than CNFs. Second, we investigate how to efficiently couple a linear programming routine with a DPLL-based SAT solver in order to obtain a solver which is tailored for BMC of hybrid systems with linear dynamics. In particular, we examine how to exploit the unique characteristics of BMC formulae and the incremental nature of BMC for various optimizations inside the solver. Finally, we present our main contribution, a tight integration of the DPLL procedure with interval constraint solving, resulting in an algorithm, called iSAT, which generalizes the DPLL procedure and is capable of solving arbitrary Boolean combinations of nonlin-

ear arithmetic constraints over the reals, even such involving transcendental functions. We demonstrate the applicability of our methods using benchmarks from the envisaged application domain.

# Zusammenfassung

Durch den zunehmenden Einsatz immer komplexerer computerbasierter Systeme in sicherheitskritischen Anwendungen gewinnt die formale Verifikation derartiger Systeme mehr und mehr an Bedeutung. Zu den erfolgreichsten Verifikationsmethoden für zustandsendliche Systeme gehört das Bounded Model Checking (BMC), eine Technik zur Prüfung der Erreichbarkeit unsicherer Systemzustände durch Abläufe mit einer festen Anzahl von Schritten. BMC gehört zu einer Gruppe von Verifikationsmethoden, die das Verifikationsproblem auf das Erfüllbarkeitsproblem einer Formel oder eine Folge solcher Probleme reduzieren. Wenngleich BMC ursprünglich für diskrete Transitionssysteme formuliert wurde, ist die Technik auch auf hybride diskret-kontinuierliche Systeme übertragbar. Letztere ergeben sich auf natürliche Weise z.B. im Bereich der eingebetteten Systeme, wo digitale (diskrete) Regler mit analogen (kontinuierlichen) physikalischen Umgebungen interagieren. Die Formeln, welche beim Bounded Model Checking derartiger Systeme entstehen, sind jedoch nicht mehr rein Boolesch, sondern beinhalten komplexe Boolesche Verknüpfungen arithmetischer Constraints über reellwertigen Variablen, deren Lösung neuartige Entscheidungsverfahren erfordert.

Die Entwicklung solcher Verfahren ist Gegenstand dieser Arbeit. Eine Schlüsselkomponente der vorzustellenden Verfahren ist die DPLL-Prozedur für das Lösen Boolescher Formeln in konjunktiver Normalform (CNF). Als ersten Beitrag zeigen wir, dass jene Optimierungen, denen der enorme Leistungszuwachs DPLL-basierter Solver in der jüngeren Vergangenheit geschuldet ist, sich für das Lösen von Systemen pseudo-Boolescher Constraints, einer im Vergleich zur CNF sehr viel konziseren Darstellungsform Boolescher Funktionen, verallgemeinern lassen. Zweitens betrachten wir die Kopplung einer Linear-Programming-Routine mit einem DPLL-basierten SAT Solver. Das Ergebnis ist ein Solver, welcher auf das Bounded Model Checking hybrider Systeme mit linearer Dynamik zugeschnitten ist. Vor diesem Hintergrund untersuchen wir insbesondere verschiedene Optimierungen, welche die besondere Struktur von BMC-Formeln sowie die inkrementelle Vorgehensweise von BMC beim

Lösen der Formeln ausnutzen. Als Hauptbeitrag stellen wir schließlich den iSAT-Algorithmus vor, welcher die DPLL-Prozedur mit Techniken des intervallbasierten Constraint-Lösens kombiniert. Der iSAT-Algorithmus ist eine Verallgemeinerung der DPLL-Prozedur und erlaubt die Prüfung der Erfüllbarkeit Boolescher Kombinationen nichtlinearer Constraints auf den reellen Zahlen, insbesondere auch von Constraints, welche transzendente Funktionen beinhalten. Wir demonstrieren die Anwendbarkeit unserer Methoden anhand von Benchmarks aus der genannten Anwendungsdomäne.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

*Humans occasionally make mistakes, even programmers do.*

*Gerard J. Holzmann*

The steadily growing complexity of computerized systems makes the latter highly susceptible to potentially critical design errors, ranging from simple programming bugs to scenarios and use cases overlooked by the engineers and therefore not properly handled by the system. The increasing use of computers in safety-critical applications like cars, aircrafts, power plants, or medical devices is therefore becoming a major concern. The risk entailed by delegating more and more safety-critical functions to computers is only acceptable if computing science manages to come up with new methods and tools which help to master the complexity of such systems and thus ensure their safety.

## 1.1 Formal Verification Using Satisfiability Checking

Simulation and testing, currently still being the most commonly practiced methods for design validation, allow to explore a relatively small number of executions of a system only and may thus fail to detect scenarios that expose a fatal bug. Except for trivial systems, simulation and testing are therefore insufficient to provide evidence for the correctness of a design.

An alternative approach for checking whether a design conforms to a desired property is formal verification. As opposed to simulation and testing, formal verification performs an exhaustive exploration of all possible behaviours of a system and is therefore guaranteed to detect existing errors or prove their absence. During the last ten years, formal verification of finite-state systems, such as digital circuits or communication protocols, has evolved from an academic subject to an approach accepted by the industry, with a number of commercial tools now being available and used by major companies.

The recent progress in SAT solving, i.e. the problem of deciding whether a given propositional formula has a solution, has motivated intensive research on how to exploit the strength of modern SAT solvers for formal verification. The result of this

C. Herde, *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure* , DOI 10.1007/978-3-8348-9949-1_1,

effort are various verification algorithms, having in common that the verification task is reduced to the problem of checking the satisfiability of a propositional formula or a series thereof.

Arguably the most successful method in SAT-based verification of discrete systems is bounded model checking (BMC), as suggested by Groote et al. in [67] and by Biere et al. in [20]. In its most simple form, BMC is used to decide whether an unsafe system state is reachable via executions whose length is bounded by some given integer $k$. More precisely, BMC looks for a run of a finite transition system which starts in an initial state of the system, obeys the system's transition relation for $k$ steps and ends in a state violating the safety property to be checked. The idea of BMC is to decide the existence of such a run, also referred to as error trace, by checking the satisfiability of the propositional formula

$$\phi_k = \bigwedge_{i=0}^{k-1} T(\vec{y}^{\,i}, \vec{y}^{\,i+1}) \ \wedge \ I(\vec{y}^{\,0}) \ \wedge \ R(\vec{y}^{\,k})$$

where $\vec{y}^{\,i}$ is the system's state-vector at step $i$ of the run, $T(\vec{y}^{\,i}, \vec{y}^{\,i+1})$ encodes the transition relation, describing all admissible state transitions between steps $i$ and $i+1$, and $I(\vec{y}^{\,0})$ and $R(\vec{y}^{\,k})$ are formulae characterizing the set of initial states and the set of unsafe states, respectively. By construction, $\phi_k$ is satisfiable if and only if an error trace of length $k$ exists. The latter is checked by translating $\phi_k$ into conjunctive normal form (CNF) and solving it with a SAT solver. The solver not only outputs a yes/no result, but in case of satisfiability also delivers a satisfying assignment which directly corresponds to an error trace, i.e. a concrete scenario illustrating how the system can enter an unsafe state, thus providing a valuable debugging aid for the designer. BMC is usually carried out incrementally by constructing and solving $\phi_k$ for increasing values of $k$ until either an error trace is found or some user specified limit on $k$, the solver's runtime, or its memory consumption is reached.

For every finite-state design there is a number $k_c$ such that non-existence of an error trace of length less or equal than $k_c$ implies that there is also no error trace of length greater than $k_c$. In theory, this indicates a way to make BMC complete, since $\phi_k$ has to be checked for $0 \leq k \leq k_c$ only. For this reason, $k_c$ is called completeness threshold. In practice, however, $k_c$ is usually too large to be reached by BMC and, moreover, cannot be computed efficiently [35]. BMC is therefore primarily seen as as method for falsification, i.e. for detecting bugs, rather than for proving their absence.

To cure the incompleteness of BMC, several techniques for *unbounded* model checking based on satisfiability checking have been proposed. Bjesse and Claessen

[24] and Sheeran et al. [109] suggest checking of safety properties using (temporal) induction schemes, where base case and induction step are encoded as Boolean formulae whose validity is checked with a SAT solver. Another approach, proposed by McMillan [89], uses Craig interpolants, which are derived from unsatisfiable BMC instances, to iteratively compute a propositional formula characterizing an overapproximation of the reachable state set of the system to be verified. To prove the safety of the system, it suffices to check whether the conjunction of this formula with a formula describing the set of unsafe states is unsatisfiable.

Albeit being limited to checking behaviours of bounded depth, BMC has proven to be a valuable tool for system validation. Its attractiveness is, amongst others, due to the fact that it has been found to scale better with the problem size than verification methods which use binary decision diagrams[1] to characterize the reachable state set [25], like classical symbolic model checking [31]. Enabled by the impressive performance gains of propositional SAT checkers in recent years, BMC can now be successfully applied even to very large finite-state designs [39].

## 1.2 Bounded Model Checking of Hybrid Systems

Often, safety-critical systems are not purely discrete, but their dynamics is characterized by an interaction of discrete and continuous behaviour. Such hybrid behaviour is found e.g. in the field of embedded systems where digital controllers are coupled with analog physical plants via sensors and actuators (see figure 1.1). Many properties of embedded controllers cannot be fully understood without taking into account the physical environment they are supposed to control. An analysis of such properties by means of formal verification requires a model of the combined hybrid discrete-continuous system.

As an example for a hybrid system consider a process control system of a chemical production process[2]. The state of the chemical plant is given by continuous variables, describing e.g. the fill level of tanks or temperature and pressure in various parts of the plant. Its evolution over time can be modeled by differential or difference equations. Each phase of the production process (e.g. evaporation of a fluid, draining a tank) corresponds to a discrete mode in the controller. Depending on the current mode, the controller opens or closes valves and activates or deactivates heaters and

[1] Binary decision diagram (BDDs) are graph-based data structures for efficient representation and manipulation of Boolean functions. For more information on BDDs, we refer the reader to [29].

[2] For concrete models of this kind see [82] and [54].

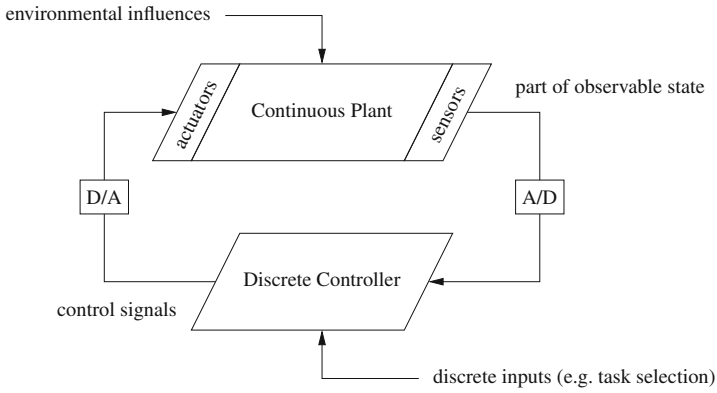


Figure 1.1: Standard set-up for a computer-controlled system.

pumps to control the respective phase of the process. Mode changes of the controller are triggered by progress of time or by state variables crossing specific thresholds (temperature of the fluid reaches boiling point, fluid level in the tank falls below a certain value). The continuous evolution of the chemical process is thus interleaved with discrete mode switches in the controller, yielding the typical discrete-continuous dynamics of a hybrid dynamical system.

However, hybrid dynamics not only arises from the use of digital control laws in a continuous physical environment. Often, a physical process itself has a natural hybrid representation. A four-stroke combustion engine, for example, is naturally modeled as a hybrid system[3]: While the four-stroke cycle of each piston can be described as a finite-state machine with four control modes, the individual phases intake, compression, expansion, and exhaust can each be modeled as a continuous process. Discrete and continuous dynamics are tightly coupled: The torque produced by each piston depends on the mode it is in. Conversely, mode transitions are triggered by the continuous dynamics of the power-train which, in turn, is driven by the engine torque.

A mathematical framework for modeling hybrid systems are hybrid automata. A hybrid automaton consists of a finite state machine where each control mode is equipped with continuous dynamics, usually specified by a set of differential equations, and a mode invariant, describing a region of the continuous state space that must not be left while staying in that mode. During a so-called flow, the continu-

[3] A detailed hybrid engine model like the one sketched here is described by Balluchi et al. in [8].

ous state changes according to the differential equations associated with the current control mode, while the discrete state (i.e. the control mode) remains unchanged. A discrete transition between control modes (also called jump) requires satisfaction of a certain guard condition and may also re-initialize continuous state variables. Guard conditions and mode invariants are usually expressed in form of arithmetic constraints over continuous state variables or inputs of the system.

Because of their relevance for industrial applications, hybrid systems have attracted a lot of interest in recent years. While tools for building hybrid models and for simulating their dynamics are commercially available (e.g. SIMULINK with the STATEFLOW extension[4]), theories and tool support for verifying hybrid systems are not yet mature. Current verification tools fall short with respect to both the dimensionality of the continuous state spaces and the size of the discrete state spaces they can handle. According to Asarin et al. [5] the *'analysis of systems with more than few continuous variables is considered very hard'*.

A promising approach to reduce the gap we observe between hybrid systems that can be specified and those that can be verified is to lift bounded model checking and related SAT-based methods for unbounded model checking to the hybrid domain. Though originally formulated for discrete transition systems only, the basic idea of BMC to reduce the search for an error path to a satisfiability problem of a formula also applies to hybrid discrete-continuous systems. However, the BMC formulae arising from such systems are no longer purely propositional. Encoding not only the discrete switching behaviour of the hybrid system, but also solution functions of differential equations, mode invariants and guard conditions, they usually comprise complex Boolean combinations of arithmetic constraints over real-valued variables, thus entailing the need for new decision procedures to solve them. The development of such procedures is the subject of this thesis.

## 1.3 Solvers for Boolean Combinations of Numerical Constraints

We address the problem of solving mixed propositional-numerical formulae like those arising as verification conditions from BMC of hybrid systems. We are thus aiming at formulae with the following characteristics:

The formulae are quantifier free and built from numerical constraints and propositional variables using arbitrary Boolean connectives. Numerical constraints are linear and nonlinear equalities and inequalites (strict or non-strict) involving variables

[4] SIMULINK and STATEFLOW are registered trademarks of The MathWorks, Inc.

of type real, integer and Boolean (the latter being interpreted as 0-1 variables when occuring inside arithmetic expressions). Besides polynomials, numerical constraints may involve transcendental functions, such as sine, cosine and the exponential function, which arise naturally e.g. in connection with solutions of differential equations. The formulae we address typically contain thousands of variables and thousands of constraints. Furthermore, they have a unique repetitive (we also say *symmetric*) structure since they comprise a $k$-fold unwinding of a transition relation.

Given such a formula, our algorithmic goals are to check the satisfiability of the formula and, in case of satisfiability, determine a sample solution, i.e. a variable assignment under which the formula evaluates to true. In the context of BMC, the sample solution will provide us with an error trace, i.e. a scenario refuting the property under verification.

### 1.3.1 Constraint Solving in a Nutshell

Solving formulae is a basic activity in mathematics and computing, yet one which is inherently difficult for many formula classes of practical relevance. From an algorithmic point of view, the problem of solving Boolean formulae and the problem of solving arithmetic constraint systems have been treated separately for a long time, resulting in specialized algorithms for the respective tasks. Solvers which combine algorithms for Boolean and numerical constraints first appeared in the 1990s.

*Boolean Formulae*

The problem of deciding the satisfiability of a Boolean formula, also referred to as SAT problem, was shown to be NP-complete by Stephen Cook in 1971 [38]. Most modern SAT solvers implement variants of the Davis-Putnam-Logemann-Loveland (DPLL) procedure which was devised already in the early 1960s [43, 42]. DPLL requires the input formula to be in conjunctive normal form (CNF), i.e. a conjunction of clauses, where a clause is a disjunction of literals, the latter being a Boolean variable or its negation. Hence, the problem has to be translated into this format first. The DPLL procedure aims at incrementally extending a partial variable assignment towards a satisfying assignment by alternately performing logical deduction, which involves an exhaustive application of the so-called unit propagation rule, and branching, i.e. tentatively assigning a truth-value to a previously unassigned variable, thus providing new input to the subsequent deduction phase. In case of a conflict, i.e. if the current partial assignment violates the input formula, DPLL backtracks to an earlier choice point and tries another branch. This is repeated until either a solution is found or backtracking

| Satisfiability | over $\mathbb{R}$ | over $\mathbb{Z}$ | over $\mathbb{N}$ | over $\mathbb{B}$ |
|---|---|---|---|---|
| Linear equations | polynomial | polynomial | NP-compl. | NP-compl. |
| Linear inequations | polynomial | NP-compl. | NP-compl. | NP-compl. |
| Polynomial equations | decidable | undecid. | undecid. | NP-compl. |
| Polynomial inequations | decidable | undecid. | undecid. | NP-compl. |
| Transcendental equations | undecid. | undecid. | undecid. | NP-compl. |
| Transcendental inequations | undecid. | undecid. | undecid. | NP-compl. |

Table 1.1: Solvability of numerical constraints.

runs out of branches in which case the formula is unsatisfiable. State-of-the-art SAT solvers enhance this basic procedure with numerous algorithmic improvements, such as lazy clause evaluation for speeding up the deduction process and non-chronological backtracking which involves an analysis of the reason for a conflict in order to back up directly to the earliest choice which caused the conflict. The latter is typically used in connection with conflict-driven learning, a technique which adds so-called conflict clauses to the input formula in order to avoid repeated failures due to the same reason. Modern SAT solvers which incorporate these enhancements are capable of solving CNFs with hundreds of thousands of Boolean variables and clauses. See [126] for a more detailed report on techniques employed in modern SAT solvers.

*Numerical Constraints*

Decision procedures for numerical constraints usually solve systems of *simultaneous* constraints, i.e. sets of constraints which are implicitly assumed to be conjoined. An extensive overview of solving algorithms for various classes of numerical constraints is given in [26]. In terms of computational complexity, the problem of checking the satisfiability of a set of simultaneous numerical constraints ranges from polynomial to undecidable, depending on the domains of the variables and the operations involved. Table 1.1 summarizes the state of affairs.

Systems of linear equations over the reals can be solved in polynomial time using Gaussian elimination. If also linear inequalities over the reals are involved, procedures for linear programming (LP) like the simplex algorithm can be used. The linear programming problem was shown to be solvable in polynomial time by Leonid Khachiyan in 1979 [80]. Although the simplex algorithm is known to have a worst-case exponential behaviour, this complexity does very rarely manifest itself in practice. Sophisticated implementations are capable of solving problems with tens of

thousands constraints and millions of variables. If some or all variables of a linear system are required to take integer values only, the complexity of deciding its satisfiability jumps from polynomial to NP-complete. Extensions of LP to mixed integer linear programming (MILP), where not all variables are required to be integer, and to pure integer linear programming (ILP) are typically limited to problems involving a few hundred or thousand integer variables. We note that CNF-SAT is a special case of solving linear systems over the integers, since each CNF clause can be translated into a linear inequality over 0-1 variables. The clause $x \vee \overline{y}$, for example, can be written as $x + (1 - y) \geq 1$.

The solvability of systems of polynomials over the reals follows from the decidability of elementary geometry which was proved by Tarski in 1948 [114]. The first decision procedure with elementary complexity, called cylindrical algebraic decomposition(CAD), was devised by Collins in 1975 [37]. The idea underlying CAD is to decompose the solution space into a finite number of so-called cells such that every polynomial has a constant sign in each cell. The worst-case number of cells, however, is doubly exponential in the number of variables occuring in the input formula such that CAD can only be applied to very small problems. Due to the undecidability of Hilbert's tenth problem [88], there is no similar solving algorithm for systems of polynomials over the integers. For the very same reason, the satisfiability of transcendental constraints over the reals is undecidable, because the periodicity of trigonometric functions can be used to filter out a model of the integers from the reals.

A more practical approach to tackle polynomial and transcendental constraints is interval constraint solving (ICS) [17]. ICS calculates a finite set of interval boxes whose union is an overapproximation of the solution set. To this end, ICS recursively refines an initial box-covering of the search space by applying interval-arithmetic deduction rules to boxes, thereby pruning parts of the interval boxes which contain non-solutions only, and by splitting boxes into sub-boxes if otherwise no further pruning can be achieved. The minimal width of a box which is selected for splitting controls the tightness of the overapproximation.

*Boolean Combinations of Numerical Constraints*

Checking the satisfiability of a propositional formula involving numerical constraints is an instance of the so-called Satisfiability Modulo Theories (SMT) problem, i.e. the problem of deciding the satisfiability of a quantifier-free formula which is a Boolean combination of propositions and atomic formulae over some background theory $T$.

Currently, the most effective approach to SMT is to combine a propositional SAT solver with a decision procedure for conjunctions of atoms from the respective background theory as follows. The SAT engine enumerates satisfying assignments of a Boolean abstraction of the input formula which is obtained by replacing each theory constraint with a new propositional variable representing the truth value of the respective constraint. A satisfying assignment of the Boolean abstraction thus defines a set of $T$-atoms that have to be satisfied simultaneously in order to satisfy the original formula. This is checked by the $T$-solver. If the set of $T$-constraints turns out to be inconsistent, an explanation for the conflict (usually a subset of $T$-constraints which causes the inconsistency) is used to refine the Boolean abstraction. The refinement process is iterated until either the Boolean abstraction (and, hence, the input formula) turns out to be unsatisfiable or a $T$-consistent satisfying Boolean assignment is found, in which case the input formula is satisfiable as well. The use of a DPLL-based SAT solver enables an optimization of the above scheme which exploits the incremental construction of an assignment in the DPLL procedure. Instead of checking only complete propositional assignments against the background theory, the $T$-consistency of a truth assignment is already verified for partial assignments, e.g. after each deduction phase of the DPLL solver. This allows an early pruning of the Boolean search tree. The resultant solver architecture is called DPLL($T$).

The solver integration sketched above is often referred to as the *lazy* approach to SMT since inconsistencies between $T$-constraints are detected and learned on the fly during DPLL proof search. As opposed to this, the *eager* approach uses the $T$-solver prior to the SAT search in order to perform a satisfiability-preserving translation of the input formula into a purely propositional formula which is then solved by a SAT solver. The eager approach, however, often suffers from a massive blow-up of the formula size during the translation. This is in particular true for arithmetic theories, which renders the eager approach impractical for input formulae containing more than a small number of theory constraints. In-depth surveys of current SMT technology are given in [12] and, with a focus on the lazy approach, in [108].

### 1.3.2 Contributions of the Thesis

The research we report in this thesis contributes to the state-of-the-art in solving mixed propositional-numerical formulae in a number of aspects. We present our contributions by addressing formula classes of increasing expressiveness, starting out with purely propositional formulae, then advancing to Boolean combinations of *linear* equalities and inequalites, and finally dealing with Boolean combinations of *nonlinear*

(and transcendental) constraints. Although being dedicated to the task of verification of hybrid systems, most of the techniques we describe are general purpose and have applications in many other domains, like operations research, planning, software verification, and scheduling, for example.

*Contribution 1: Acceleration of DPLL for Pseudo-Boolean Constraints*

A key component in the solving algorithms we present in this thesis is the DPLL procedure for checking CNF satisfiability. The algorithms we propose extend and generalize the DPLL procedure. Hence, any improvement made to DPLL directly carries over to the performance of the solvers for mixed propositional-numerical formulae. Our first contribution therefore addresses, as a start, propositional satisfiability and investigates the problem of generalizing acceleration techniques as found in recent satisfiability engines for conjunctive normal forms (CNFs) to systems of linear constraints over the Booleans, also called pseudo-Boolean constraints. The motivation for this research is that compared to clauses of a CNF, pseudo-Boolean constraints allow a significantly more compact description of many discrete problems. Their modeling power has been widely used in the fields of logic synthesis, operations research, and formal verification, see [3] and [30] for example. We demonstrate that acceleration techniques like lazy clause evaluation, non-chronological backtracking and learning techniques generalize smoothly to Davis-Putnam-like procedures for the very concise propositional logic of linear constraint systems over the Booleans. Despite the more expressive input language, the performance of our prototype implementation comes surprisingly close to that of CNF-SAT engines like CHAFF [91]. Experiments with bounded model-construction problems show that the overhead in the satisfiability engine that can be attributed to the richer input language is often amortized by the conciseness gained in the propositional encoding of the BMC problem.

*Contribution 2: BMC-Specific Optimizations in a DPLL(LA) Solver*

As a second step, we address the problem of solving Boolean combinations of constraints from the theory of linear arithmetic over the reals (LA). This subclass of formulae is especially appealing because it is decidable, i.e. allows for a complete solving algorithm, and it is powerful enough to model hybrid automata with linear dynamics. The latter are of particular practical relevance since nonlinear physical systems can often be approximated with sufficient accuracy by piecewise linear approximations. We aim at providing a solver which is tailored for BMC of linear hybrid automata. To this end, we propose a DPLL($T$) architecture where the $T$-solver is instantiated with

a linear programming (LP) routine which is used to decide the feasibility of sets of linear constraints. We investigate how to take advantage of the specific properties of linear programming to make the coupling of DPLL and LP as efficient as possible. In particular, we study various methods for computing small (ideally minimal) infeasible subsystems of a conflicting linear constraint system which serve as explanations for arithmetic conflicts. Explanations are learned by the SAT solver in form of conflict clauses which are used to resolve the conflict and in order to prevent the solver from running into further arithmetic conflicts due to the same reason. We examine empirically the impact of these methods on the runtime of the DPLL(LA) engine. Furthermore, and most importantly, we exploit the unique characteristics of BMC formulae (in particular their symmetry) and the incremental nature of BMC for a variety of optimizations in the solver. We demonstrate that these optimization, which were originally proposed by Ofer Strichman for BMC of purely discrete systems [113], pay off even better in the hybrid domain. The reason is that an inference step involving arithmetic constraints is computationally much more expensive than one involving propositional logic only. To the best of our knowledge, our solver was the first to make use of BMC-specific optimizations in order to accelerate solving of formulae arising from verification of hybrid systems.

*Contribution 3: Integration of DPLL and Interval Constraint Solving*

Finally, we address the problem of solving large Boolean combinations of nonlinear arithmetic constraints involving transcendental functions. For this purpose, we propose a tight integration of a DPLL-based SAT solver and interval-arithmetic constraint solving. The undecidability of the arithmetic base theory, however, precludes the use of a DPLL($T$) approach since DPLL($T$) heavily relies on the availability of a *complete* theory solver. Instead, we exploit the algorithmic similarities between DPLL-based propositional SAT solving and interval constraint solving for a much tighter integration, where the DPLL solver directly controls arithmetic constraint propagation rather than delegating arithmetic inferences to a subordinated theory solver. The resulting solving algorithm, which we refer to as iSAT, turns out to be an elegant generalization of the DPLL routine. It inherits the branch-and-deduce framework and the unit propagation rule for logical deductions from DPLL and adds deduction rules for arithmetic operators which are adopted from interval constraint solving. Through this tight integration, all the algorithmic enhancements that were instrumental to the enormous performance gains recently achieved in propositional SAT solving carry over smoothly to the rich domain of nonlinear arithmetic constraints. We demonstrate that

our approach is able to handle large constraint systems with complex Boolean structure, involving Boolean combinations of multiple thousand arithmetic constraints over some thousands of variables. Having the structure of a DPLL routine, the ISAT algorithm can be extended with all algorithmic contributions discussed in the previous sections. It can be tuned for BMC using Strichman's optimizations, it can be equipped with a special deduction rule for pseudo-Boolean constraints, and a lazy integration of linear programming, yielding an ISAT (LA) solver, will strengthen ISAT's deductive power for linear constraints. We consider the ISAT algorithm to be the main contribution of this thesis.

### 1.3.3 Structure

Subsequent to this introduction, chapter 2 sets out to give some insight into the origin of the formulae we are concerned with. After recalling some basic definitions on hybrid automata, we demonstrate by means of a running example various ways to encode the next-state relation of a hybrid automaton as a formula involving Boolean and arithmetic constraints.

The following three chapters present the contributions explicated above. Chapter 3 deals with extending DPLL for pseudo-Boolean constraints, chapter 4 investigates the coupling of DPLL with linear programming and tuning the resulting solver for BMC, and chapter 5 describes the ISAT algorithm for solving Boolean combinations of nonlinear constraints. Each chapter first defines the formula class to be tackled and thereafter reviews the preliminaries which are required to prepare the reader for the subsequent exposition of the algorithmic contributions. A final section reports on the benchmarks conducted to evaluate specific aspects and the overall performance of the methods proposed in the respective chapter. In particular, a case study of a controller for separation of trains being operated under a moving block principle is presented in chapter 5 in order to demonstrate the applicability of the ISAT approach to designs from the envisaged application domain.

Chapter 6 provides a summary of the results of the thesis, followed by suggestions for future research.

### 1.3.4 Sources

This thesis draws from the following conference and journal publications of the author. Chapter 3 is based on a conference paper published at LPAR'03 [60]. The article which forms the basis of chapter 4 received the EASST best paper award at FMICS'04

and was published in *Electronic Notes of Theoretical Computer Science* in 2005 [61]. A blend of material from [60] and [61] appeared 2007 in *Formal Methods in Systems Design* [62]. Chapter 5 is a revised and extended version of an article published in 2007 in the *Journal on Satisfiability, Boolean Modeling and Computation* [63]. The train case study presented in chapter 5 was published at ICONS'08 [74].

The contributions and achievements of this thesis, though original work by myself, have obviously been influenced and shaped in many discussions with the co-authors of the above papers and many other researchers. In particular, I benefited a lot from the exchange and tight collaboration with friends and colleagues from the *University of Freiburg*, *Saarland University*, the *Academy of Sciences of the Czech Republic*, and the *University of Oldenburg* within the Transregional Collaborative Research Center 'Automatic Verification and Analysis of Complex Systems' (AVACS), funded by the DFG.

# 2 Hybrid Dynamical Systems

We aim at providing solver technology to be used in tools for formal verification of hybrid dynamical systems. The verification methods we are going to support require a precise mathematical model of the system under investigation in order to be applicable.

The use of mathematical models has a long tradition in science and engineering, since models allow the study of systems without actually constructing and operating the latter, which might be impractical, expensive, time-consuming or even hazardous. Mathematical modeling is a challenging task which requires a thorough understanding of the system under consideration. The development of a model which is suitable for verification is even more demanding because the model must not only adequately capture those aspects of the system which are relevant w.r.t. the properties to be checked, but at the same time be simple enough to not go beyond the capabilities of the verification methods to be applied.

The purpose of this chapter is to demonstrate how the dynamics of a hybrid system can be encoded as a predicative formula and thereby to shed light onto the origins of the specific characteristics of the formulae we have to deal with in the solving engines. We intend this to be background information for the reader and do not claim originality of the material presented here.

As modeling formalism for hybrid systems we use hybrid automata, which have proven to be a useful framework for theoretical reasoning about hybrid systems, e.g. for investigation of questions like decidability [73]. We opt for hybrid automata (ruling out more practically-oriented languages like SIMULINK) because of their simple and well-defined semantics. The principle ideas underlying our encondings are, however, easily transferable to different, potentially syntactically richer modeling languages.[1]

[1] See section 5.5, page 119 for the translation of a hybrid system modeled in the SIMULINK language into a predicative formula.

C. Herde, *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure*, DOI 10.1007/978-3-8348-9949-1_2,

In section 2.1 we briefly recap hybrid automata as a modeling framework for hybrid systems and define their semantics. To illustrate the definitions and concepts, we present a simple hybrid automaton model of a household freezer. Though being a toy example, the freezer model shares many typical properties of more realistic hybrid systems: It involves nondeterministic choices which are used to model uncontrollable environment behaviour, nonlinear continuous dynamics yielding trajectories whose description requires transcendental functions, and it is complex enough to give raise to a nontrivial verification task.

We re-use this model in section 2.2 to explain various encodings of a hybrid automaton as a logical formula. In particular, we deal with different representations of nonlinear flows in the BMC formulae. Using one such encoding, we demonstrate how to solve the aforementioned verification task by bounded model checking. While encodings very similar to ours have been used by Audemard et al. [7] and Bemporad et al. [15] for BMC of linear hybrid automata, we are not aware of of related work on nonlinear BMC encodings. The latter can probably be attributed to the lack of decent solvers for this domain.

## 2.1 Modeling Hybrid Systems with Hybrid Automata

A hybrid system is a special form of a dynamical system, i.e. a system which is characterized by a system state and a dynamical rule which describes the evolution of the state over time. It can be equipped with inputs and outputs for interaction with the environment. At any point in time, the system state provides a description of the system which together with the system's current input is sufficient to determine the next state of the system. In particular, no recourse to states prior to the present state is required for the computation of the next state. The state is described by the valuation of a set of state variables and can thus be interpreted as a point in the geometric space defined by the Cartesian product of those variables, the state space. Typically, but not necessarily, a hybrid system has not only real-valued (i.e. continuous) state variables, but also discrete ones (i.e. state variables of type Boolean or integer). The dynamical rule not only allows (Lipschitz-) continuous evolutions of the state (so-called flows), but also discrete transitions (called jumps) between distant points in the state space.

In the following definitions (and throughout this thesis), we use arithmetic predicates and Boolean combinations thereof to describe sets of system states.

**Definition 2.1.** Let $Y = \{y_1, \ldots, y_n\}$ be the set of state variables of a dynamical system.

- A *valuation* or *state* is a mapping that assigns to each variable $y \in Y$ a value from its domain $\mathrm{dom}(y) \subseteq \mathbb{R}$. For notational convenience, we identify valuations with points in the state space $\Sigma = \mathrm{dom}(y_1) \times \ldots \times \mathrm{dom}(y_n)$.

- An *arithmetic predicate* or *constraint* over $Y$ is a Boolean-valued function of the form $\theta \sim c$, where $c \in \mathbb{R}$ is a constant, $\sim \in \{<, \leq, =, \geq, >\}$ is a relational operator, and $\theta$ is a term built from variables in $V$, constants in $\mathbb{R}$, and arithmetic operations (such as addition, multiplication, exponentiation, and trigonometric functions). The two 0-ary predicates are, as usual, denoted by `true` and `false`. An arithmetic predicate $\theta \sim c$ is called *linear* if $\theta$ is of the form $c_1 y_1 + \ldots + c_n y_n$, where the $c_i$ are rational numbers.

- A *predicative formula* or *predicate* over $Y$ is built from arithmetic predicates over $Y$ using the standard Boolean connectives $\neg, \wedge, \vee, \rightarrow$, and $\leftrightarrow$.

- Let $\sigma$ be a state and $\phi$ a predicative formula. We say that $\sigma$ *satisfies* $\phi$, written as $\sigma \models \phi$, if $\phi$ evaluates to true when each $y \in Y$ occuring in $\phi$ is replaced by the value $\sigma(y)$. Then $\sigma$ is called *solution* of $\phi$. The set of states defined by $\phi$ is $[\![\phi]\!] := \{\sigma \mid \sigma \in \Sigma \text{ and } \sigma \models \phi\}$. In particular, $[\![\texttt{true}]\!] = \Sigma$ and $[\![\texttt{false}]\!] = \varnothing$.

A hybrid automaton models a hybrid system as a graph whose edges describe discrete transitions and whose nodes (called control modes or locations) represent continuous flows. Discrete transitions are assumed to be instantaneous, i.e. they happen in zero time, while during a flow time elapses. Formally we define:

**Definition 2.2.** A *hybrid automaton* $H = (X, V, E, inv, init, jump, flow)$ consists of

- a finite set $X = \{x_1, \ldots, x_n\}$ of continuous (real-valued) state components,

- a finite set $V = \{v_1, \ldots, v_m\}$ of discrete control modes,

- a finite set $E$ of discrete transitions, together with mappings

  - $s : E \rightarrow V$, assigning to each transition $e \in E$ its source mode $s_e$
  - $p : E \rightarrow V$, assigning to each transition $e \in E$ its target mode $p_e$

- a family $inv = (inv_v)_{v \in V}$ assigning to each control mode $v \in V$ an invariant $inv_v$ which is a predicate over the variables in $X$,
- a family $init = (init_v)_{v \in V}$ of initial state predicates, where $init_v$ is a predicate over $X$ which specifies for control mode $v$ the admissible initial valuations of the continuous state components,
- a family $jump = (jump_e)_{e \in E}$ assigning to each discrete transition a jump condition which is defined by means of a predicate over variables in $X \cup X'$, where $X' = \{x'_1, \ldots, x'_n\}$ denotes primed variants of the state components in $X$,
- a family $flow = (flow_v)_{v \in V}$ assigning to each control mode a predicate over variables in $X \cup \dot{X}$, where $\dot{X} = \{\dot{x}_1, \ldots, \dot{x}_n\} = \{\frac{dx_1}{dt}, \ldots, \frac{dx_n}{dt}\}$ is the set of the first derivatives of the continuous state components in $X$ with respect to time.

The *(hybrid) state* of $H$ is a point $(v, z) \in V \times \mathbb{R}^n$ consisting of the current mode $v$ and the current valuation $z$ of the continuous state components of $H$. We refer to $v$ as the *discrete state* and to $z$ as the *continuous state* of $H$. At any time, the continuous state $z$ must satisfy the mode invariant $inv_v$ of the current control mode $v$. A state $(v, z)$ is *initial* if the predicate $init_v$ is satisfied by $z$. Control modes whose initial state predicate is `false` may not be taken initially.

The jump condition describes a pre-post-relation, where undecorated state components $x \in X$ refer to the state immediately before the jump, while the primed variant $x' \in X'$ refers to the state immediately thereafter. For a transition $e \in E$, the predicate $jump_e$ is a conjunction of a guard and a (potentially nondeterministic) assignment. The guard is a predicate over $X$ which specifies the continuous states that enable a jump from mode $s_e$ to mode $p_e$. The assignment is a predicate over $X \cup X'$ which relates the values of the continuous state components before the jump to possible values thereafter.

The flow predicate $flow_v$ of a mode $v \in V$ describes the evolution of the continuous state while residing in $v$. Usually it has the form $\frac{d\vec{x}}{dt} = f_v(\vec{x})$, where $\vec{x} = (x_1, \ldots, x_n)$ is the continuous state vector, i.e. it is given as a system of first order differential equations.

Graphically, we depict a hybrid automaton as a directed graph with nodes $V$ and edges $\{(s_e, p_e) \mid e \in E\} \subseteq V \times V$. Each node $v \in V$ is annotated with its flow predicate $flow_v$ and its mode invariant $inv_v$. A node $v$ that can be taken initially is marked with an incoming edge which has no source node and which is labelled with

$init_v$. Each edge $(s_e, p_e)$ is annotated with its jump condition $jump_e$. Mode invariants and guards which equal `true` (and are therefore satisfied by *any* continuous state), as well as assignments of the form $x' = x$, which do not change the value of the respective state component, are suppressed.

**Example 2.3.** We consider a simple hybrid model of a household freezer. The temperature $T_{\mathcal{I}}$ in the freezer cell evolves according to the differential equation $\dot{T}_{\mathcal{I}} = \eta \cdot (T_{\mathcal{E}} - T_{\mathcal{I}}) + c$, where $T_{\mathcal{E}}$ is the temperature of the environment, $\eta$ is the heat transfer coefficient, and $c$ models the cooling capacity of the compressor. In consideration of the slow dynamics of the system, we choose one hour (instead of one second) as time unit. Hence, the unit of $\eta$ is $\left[\frac{1}{h}\right]$, and the unit of $c$ is $\left[\frac{^\circ C}{h}\right]$. We assume $T_{\mathcal{E}}$ to be fixed at $20^\circ C$, $\eta = 0.3$ if the door of the freezer is closed, and $\eta = 2.2$ if the door is open. If the compressor is running then $c = -14$, else $c = 0$. The freezer is controlled by a conventional hysteresis controller: The compressor is switched on if $T_{\mathcal{I}}$ reaches an upper threshold of $-16^\circ C$, and it is switched off if $T_{\mathcal{I}}$ drops below a lower threshold of $-20^\circ C$. To prevent overheating, the compressor is automatically switched off when the door is open. We assume that initially the door is shut.

Figure 2.1 on the following page shows a hybrid automaton model of the freezer which has three discrete modes $V = \{\text{ON}, \text{OFF}, \text{OPEN}\}$ and a single continuous state component $X = \{T_{\mathcal{I}}\}$. The set of initial states is $\{(\text{OFF}, z) \mid z \leq -16\} \cup \{(\text{ON}, z) \mid z \geq -20\}$. At any time the door of the freezer can be opened, which is modeled by unconditional transitions from OFF to OPEN and from ON to OPEN, respectively. The system can stay in mode OPEN, where $T_{\mathcal{I}}$ evolves as determined by the differential equation $\dot{T}_{\mathcal{I}} = -2.2\, T_{\mathcal{I}} + 44$, arbitrarily long, since $inv_{\text{OPEN}} = \texttt{true}$. It can jump, for example, from mode OPEN to mode OFF along the edge (OPEN, OFF) if the continuous state satisfies the guard predicate $T_{\mathcal{I}} \leq -16$. If initially, or when being in mode OPEN, $-20 \leq T_{\mathcal{I}} \leq -16$ holds, then the model can nondeterministically choose whether to enter mode OFF or mode ON. No jump can change the continuous state, i.e. all assignments have the form $T'_{\mathcal{I}} = T_{\mathcal{I}}$ and are therefore omitted in the graphical representation.

A hybrid automaton engages in so-called executions.[2] Intuitively, an execution is a (not necessarily alternating) sequence of jumps and flows which is consistent with the dynamics defined by the automaton. Hybrid systems usually model so-called reactive systems, i.e. systems which continuously interact with an environment by reacting to external disturbances or inputs, and by producing outputs that can be perceived by the

[2] Occasionally, we also use the terms run, trajectory, trace, and path instead of execution.

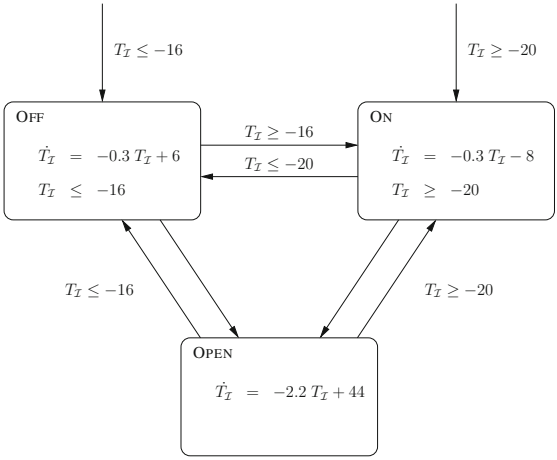


Figure 2.1: Hybrid automaton model of a household freezer.

environment. They therefore engage, at least in principle, in executions which have infinite length. Nonetheless, the following definitions will only consider executions of finite length, since these are the only ones we will encounter in practice when doing bounded model checking. To prepare a formal definition of executions, we first define the concept of a hybrid time frame.

**Definition 2.4.** A *hybrid time frame* is a mapping $\tau : \{0, \ldots, k\} \to \mathbb{R}_{\geq 0}$ with $\tau(0) = 0$ and $\tau(i) \leq \tau(i+1)$ for $0 \leq i < k$. We write $\tau_i$ instead of $\tau(i)$ and identify $\tau_i'$ with $\tau_{i+1}$. The set of all hybrid time frames is denoted by $\mathcal{T}$.

The indices $\{0, \ldots, k\}$, which we refer to as *steps* of the hybrid time frame, consecutively number states within an execution which are starting points and endpoints of flows and jumps. The values $\tau_i$ are the points in time at which the execution traverses these states. Since a jump is instantaneous, a pair of indices $(i, i+1)$ with $\tau_i = \tau_{i+1}$ denotes a jump. If $\tau_i < \tau_{i+1}$, it represents a flow. We refer to $k$ as *length* of a hybrid time frame, and to $\tau_k$ as its *duration*.

The following definition of an execution associates with each step $i$ of a hybrid time frame two partially defined functions $v_i$ and $x_i$ which assign a discrete and a continuous state to the time instance $\tau_i$, and which specify the evolution of those states over the time interval $[\tau_i, \tau_i']$ in case that $i$ denotes the starting point of a flow.

**Definition 2.5.** Let $H = (X, V, E, inv, init, jump, flow)$ be a hybrid automaton. An *execution* of $H$ is a collection $\mathcal{E} = (\tau, (v_n), (\vec{x}_n))$, where $\tau \in \mathcal{T}$ is a hybrid time frame of length $k$, $(v_n)$ is a sequence $v : \{0, \ldots, k\} \to (\mathbb{R}_{\geq 0} \xrightarrow{\text{part.}} V)$ with $\mathrm{dom}(v_i) = [\tau_i, \tau_i']$, and $(\vec{x}_n)$ is a sequence $\vec{x} : \{0, \ldots, k\} \to (\mathbb{R}_{\geq 0} \xrightarrow{\text{part.}} \mathbb{R}^n)$ with $\mathrm{dom}(\vec{x}_i) = [\tau_i, \tau_i']$ such that the following conditions are satisfied:

- $\mathcal{E}$ is grounded in an admissible initial state:

  $$\vec{x}_0(\tau_0) \models init_{v_0(\tau_0)} \text{ and}$$
  $$\vec{x}_0(\tau_0) \models inv_{v_0(\tau_0)}.$$

- For each $i \in \{0, \ldots, k-1\}$ with $\tau_i = \tau_i'$ there exists a transition $e \in \mathcal{E}$ with $s_e = v_i(\tau_i)$, and $p_e = v_{i+1}(\tau_i')$ such that
  - the pair of continuous states $(\vec{x}_i(\tau_i),\ \vec{x}_{i+1}(\tau_i')) \in \mathbb{R}^n \times \mathbb{R}^n$ is an element of the relation induced by the jump condition $jump_e$, and
  - pre- and post-state of the jump satisfy the respective mode invariant, i.e. $\vec{x}_i(\tau_i) \models inv_{v_i(\tau_i)}$ and $\vec{x}_{i+1}(\tau_i') \models inv_{v_{i+1}(\tau_i')}$.

- For each $i \in \{0, \ldots, k-1\}$ with $\tau_i < \tau_i'$
  - $v_i$ is a constant function which satisfies $v_{i+1}(\tau_i') = v_i(\tau_i')$, and
  - $\vec{x}_i(t)$ is a solution to the differential equations $\frac{d\vec{x}}{dt} = f_{v_i(\tau_i)}(\vec{x})$ which preserves the invariant of the current mode, i.e. $\vec{x}_i(t) \models inv_{v_i(\tau_i)}$ for each $t \in [\tau_i, \tau_i']$, and which satisfies $\vec{x}_{i+1}(\tau_i') = \vec{x}_i(\tau_i')$.

We lift the notions *length*, *duration*, and *step* from the hybrid time frame underlying an execution to the execution itself.

**Example 2.6.** Figure 2.2 shows an execution of the freezer automaton depicted on page 20. It starts in state $(\textsc{On}, 20)$, covers a duration of $24$ hours, and has a length of $29$ steps. The upper chart shows the evolution of the continuous state, i.e. of the temperature in the freezer cell. The three charts below show the switching behaviour of the automaton: A function value of $1$ indicates that control resides in the respective mode. After $12.3$ hours, the door of the freezer is opened for $30$ minutes, causing a steep increase of the temperature. The numbers on the topmost axis denote the steps of the execution. For example, the jump of the automaton triggered by opening the door starts at step $13$ and ends at $14$. Both steps are associated with the same point in time: $\tau_{13} = \tau_{14} = 12.3$.

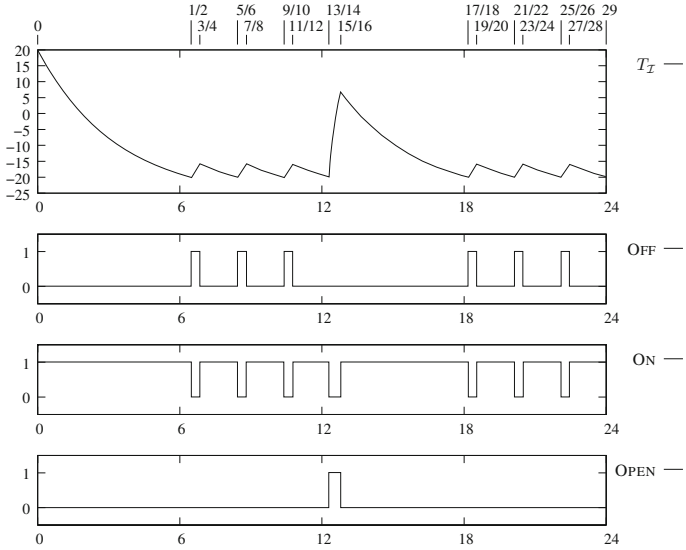


Figure 2.2: Sample execution of the freezer automaton.

We can now give a formal definition of reachability:

**Definition 2.7.** A state $(v, z) \in V \times \mathbb{R}^n$ is *reachable* by a hybrid automaton $H = (X, V, E, inv, init, jump, flow)$, if there exists an execution $\mathcal{E} = (\tau, (v_n), (\vec{x}_n))$ of $H$ with length $k$ such that $(v_k(\tau_k), \vec{x}_k(\tau_k)) = (v, z)$.

The notion of reachability is closely connected with safety verification, because proving a safety property amounts to showing that the intersection of the reachable state set with the set of unsafe system states is empty. Many verifications problems can be reduced to checking the reachability of certain states.

**Example 2.8.** For the freezer automaton we wish to verify the claim that $T_{\mathcal{I}}$ will never exceed $0°C$ if initially $-20° \leq T_{\mathcal{I}} \leq -16°C$, provided that the door is never opened for more than three minutes and remains closed thereafter for at least twice the opening time before it's opened again. To this end, we modify the hybrid automaton from figure 2.1 in order to restrict its executions to such which obey the above policy for opening and closing the freezer door. We equip the automaton with an additional continuous variable $p$ which is used as a timer, i.e. it evolves with slope $1$ in each mode. Enforced by the new invariant $p \leq \frac{3}{60}$, mode OPEN is left at latest after $3$ minutes via

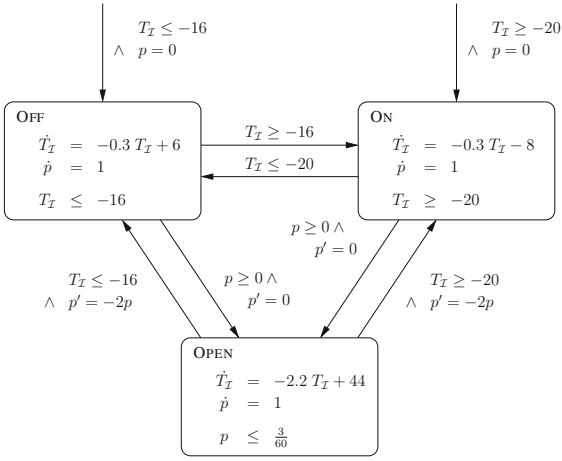


Figure 2.3: Freezer automaton, modified for verification.

a jump which resets $p$ to $-2$ times its current value. Because of the transition guards $p \geq 0$, mode OPEN cannot be re-entered until the accumulated time spent in modes ON and OFF equals twice the time that has been spent in mode OPEN before. The modified automaton, which is depicted in figure 2.3, allows us to formulate the above verification task as a reachability problem: The alleged property holds if none of the states $\{(v, z) \in V \times \mathbb{R} \mid z \geq 0\}$ is reachable from an initial state where $T_{\mathcal{I}} = -18$ holds.

## 2.2 Predicative Encodings of Hybrid Automata

In order to perform bounded model checking of hybrid automata, i.e. checking the reachability of certain 'unsafe' states through executions of finite length, we employ a reduction to SMT which encodes all executions of a given length $k \in \mathbb{N}$ as a predicative formula. There are various ways of doing this, all with specific strengths and weaknesses, and especially for systems with nonlinear dynamics the translation can be a non-trivial task.

For a given hybrid automaton and a given $k \in \mathbb{N}$ we aim at constructing a formula such that each execution of the automaton which ends in an unsafe state corresponds

to a satisfying valuation of the formula. Ideally, also the converse holds, so that satisfying valuations are in one-to-one correspondence with error traces. Some predicative encoding schemes are, however, based on building safe approximations[3] of the system dynamics. The resulting BMC formulae can have solutions which translate back to spurious error traces, i.e. traces which do not reflect actual system behaviour. On the other hand they are safe w.r.t. verification, since unsatisfiability of such formulae implies the non-existence of error traces of length $k$.

In the following, we will only deal with encodings of a single hybrid automaton. Using standard techniques from predicative semantics [71], the translation scheme can, however, be extended to both shared variable and synchronous message passing parallelism, thereby yielding formulae of size linear in the number of parallel components. The latter is due to the fact that a BMC formula for a parallel system is essentially the conjunction of the formulae for the individual components. This highlights one of the main advantages of the BMC approach to hybrid systems verification: By representing the continuous and the discrete state space of a system both symbolically, BMC avoids the state-explosion problem arising from the construction of the global product automaton, a step which is inevitable for tools like HYTECH [72] and PHAVER [64], which use an explicit representation of the discrete state space.

### 2.2.1 A Basic Encoding Scheme

Let $H = (X, V, E, inv, init, jump, flow)$ be a hybrid automaton. For the encoding scheme to be presented in this section, we assume that for each control mode $v \in V$ of $H$ a closed-form solution to the differential equations $\frac{d\vec{x}}{dt} = f_v(\vec{x}\,)$ pertaining to that mode is available. The solution is a predicate of the form $\vec{x}\,(t) = F_v(t, \vec{x}\,(0))$ which relates the continuous state $\vec{x}\,(0)$ at the time point of entering $v$ to the continuous state $\vec{x}\,(t)$ after residing $t$ time units in $v$. It is closed in the sense that it can be represented explicitly in terms of known functions. In order to encode the executions of $H$ of a given length $k \in \mathbb{N}$, we proceed as follows:

*Encoding Step 1*

> For each continuous state component $x \in X$, we take $k+1$ real-valued variables $x^0, \ldots, x^k$. Variable $x^i$ encodes the value of $x$ at step $i$ of the execution. Using vector notation, the continuous state $\vec{x} = (x_1, \ldots, x_n)$ of the automaton at step $i$ is represented in the encoding by $\vec{x}^{\,i} = (x_1^i, \ldots, x_n^i)$.

[3] Safe approximations are also referred to as conservative approximations or overapproximations.

*Encoding Step 2*

For each discrete control mode $v \in V$ we take $k+1$ Boolean variables $v^0, \ldots, v^k$. Variable $v^i$ takes value `true` iff at step $i$ of the execution control resides in mode $v$. Hence, the vector $\vec{v}^{\,i} = (v_1^i, \ldots, v_m^i)$ represents the discrete state $v \in V$ of $H$ at step $i$. Through

$$\bigwedge_{i=0}^{k} \left( \sum_{v \in V} v^i = 1 \right)$$

we express, that at any time the automaton is exactly in one control mode. Here, as throughout this thesis, we identify Boolean variables with 0-1 integer variables which enables their use within arithmetic constraints.

*Encoding Step 3*

We introduce $k$ Boolean variables $j^0, \ldots j^{k-1}$. Variable $j^i$ takes value `true` if step $i$ of the execution is starting point of a jump, and `false` if it is starting point of a flow. Furthermore, we take $k$ real-valued variables $t^0, \ldots, t^{k-1}$, where the value of $t^i$ equals the time elapsing between execution steps $i$ and $i+1$. By adding the constraints

$$\bigwedge_{i=0}^{k-1} \big(j^i \rightarrow (t^i = 0)\big) \ \wedge \ \big(\neg j^i \rightarrow (t^i > 0)\big)$$

we encode that jumps are instantaneous, whereas flows require passage of physical time.

*Encoding Step 4*

A jump requires that source mode and target mode of the jump are connected by a control switch. Moreover, the continuous state before the jump and the continuous state after the jump must satisfy the jump condition of the respective control switch. This can be expressed by

$$\bigwedge_{i=0}^{k} j^i \rightarrow \bigvee_{e \in E} s_e^i \wedge p_e^{i+1} \wedge jump_e[\vec{x}^{\,i}/\vec{x}][\vec{x}^{\,i+1}/\vec{x}\,'] \wedge inv_{s_e}[\vec{x}^{\,i}/\vec{x}] \wedge inv_{p_e}[\vec{x}^{\,i+1}/\vec{x}].$$

*Encoding Step 5*

The effect of continuous flows on the state is encoded by the constraint system

$$\bigwedge_{i=0}^{k} \neg j^i \rightarrow \bigvee_{v \in V} v^i \ \wedge \ v^{i+1} \ \wedge \ \vec{x}^{\,i+1} = F_v(t^i, \vec{x}^{\,i}) \ \wedge \ inv_v[\vec{x}^{\,i}/\vec{x}] \ \wedge \ inv_v[\vec{x}^{\,i+1}/\vec{x}]$$

which expresses that a flow starts and ends in the same control mode, that the inital state of the flow is connected to its final state by a solution to the differential equations of that mode, and that inital and final state both satisfy the modes' invariant condition.

*Encoding Step 6*

In order to restrict the executions to those starting in an initial state of $H$, we add the following constraints:

$$\bigwedge_{v \in V} v^0 \rightarrow init_v[\vec{x}^{\,0}/\vec{x}]$$

*Encoding Step 7*

Finally, we complete the BMC formula by adding a predicate over $\{v_1^k, \ldots, v_m^k\} \cup \{x_1^k, \ldots, x_n^k\}$ which specifies the unsafe hybrid states whose reachability we want to check. By conjoining it to the formula, we confine the set of executions to those which end in one of the states in question.

The conjunction of the constraints generated by the above translation scheme yields a predicative formula $\phi_k$ whose external structure is identical with the one of the formula for finite-state BMC given on page 2:

$$\phi_k = \bigwedge_{i=0}^{k-1} T(\vec{y}^{\,i}, \vec{y}^{\,i+1}) \ \wedge \ I(\vec{y}^{\,0}) \ \wedge \ R(\vec{y}^{\,k})$$

The $k$-fold unwinding of the transition relation $T(\vec{y}^{\,i}, \vec{y}^{\,i+1})$ is built by encoding steps 1 to 5, and the subformulae $I(\vec{y}^{\,0})$ and $R(\vec{y}^{\,k})$, which characterize the set of initial states and the set of unsafe states, are generated by encoding steps 6 and 7, respectively. The hybrid state of $H$ at step $i$ of an execution is represented by the vector $\vec{y}^{\,i} = (x_1^i, \ldots, x_n^i, v_1^i, \ldots, v_m^i)$ which contains all variables created in steps 1 and 2.

The construction of $\phi_k$ guarantees that each execution $\mathcal{E} = (\tau, (v_n), (\vec{x}_n))$ of $H$ which has length $k$ and ends in an unsafe state corresponds to a solution $\sigma$ of $\phi_k$. Given $\sigma$, we can reconstruct $\mathcal{E}$ as follows: The hybrid time frame $\tau$ of $\mathcal{E}$ is defined by

$$\tau_i = \begin{cases} 0 & : \quad i = 0 \\ \sum_{j=0}^{i-1} \sigma(t^j) & : \quad 1 \leq i \leq k. \end{cases}$$

The valuation of the variables $v_1^i, \ldots, v_m^i$ and $x_1^i, \ldots, x_n^i$ provides the discrete state $v(\tau_i)$ and the continuous state $\vec{x}(\tau_i)$ of $H$ at step $i$ of $\mathcal{E}$. If $\sigma(j^i)$ equals $true$, then a jump starts at step $i$. Otherwise, step $i$ of the execution is starting point of a flow whose duration is $\sigma(t^i)$. While $\sigma$ only provides the initial and the final continuous state of a flow, intermediate states can be easily obtained from the solution predicate of the differential equations describing the flow.

Note that, except for step 6 and 7 of the encoding scheme, all steps generate multiple copies of the same basic formula, where the $k$ or $k+1$ individual copies differ just in a consistent renaming of the variables. Therefore, a satisfiability checker tailored towards BMC of hybrid automata should exploit such isomorphies between subformulae for accelerating satisfiability checking, which is a distinguishing feature of the solvers presented in chapter 4 and 5. In order to simplify detection of isomorphic copies, those solvers are in fact fed with just a single copy of the transition and evolution predicates and perform the unrolling themselves.

**Example 2.9.** In order to verify the property stated in example 2.8 on page 22, we encode the hybrid automaton depicted in figure 2.3 using the above encoding scheme.

To represent the state of the freezer automaton at step $i$ of an execution, we use the vector $\vec{y}^{\,i} = (T_{\mathcal{I}}^i, p^i, \mathit{off}^i, \mathit{on}^i, \mathit{open}^i)$, where the variables $T_{\mathcal{I}}^i$ and $p^i$ are of type real and $\mathit{off}^i, \mathit{on}^i$, and $\mathit{open}^i$ are of type Boolean. In addition, we use real-valued variables $t^i$ and Boolean variables $j^i$, as required by encoding step 3.

Instead of writing down the unwound transition relation of the hybrid automaton in full, we only record the constraints describing a single step of the transition relation, i.e. we leave out the outer conjunction in the constraints generated by encoding steps 2 to 5 and use unprimed and primed variables in place of variables with superscripts $i$ and $i+1$. The closed-form solutions to $\frac{dT_{\mathcal{I}}}{dt} = \eta \cdot (T_{\mathcal{E}} - T_{\mathcal{I}}) + c$ and $\frac{dp}{dt} = 1$, which we need in encoding step 5, are $T_{\mathcal{I}}(t) = (T_{\mathcal{I}}(0) - b)e^{\eta \cdot t} + b$, where $b := T_{\mathcal{E}} + \frac{c}{\eta}$, and $p(t) = p(0) + t$, respectively. Then the constraint system which encodes the transition relation looks as follows:

$$
\begin{aligned}
T(\vec{y}, \vec{y}') := \quad & \mathit{off} + \mathit{on} + \mathit{open} = 1 \\
\wedge \quad & j \rightarrow (t = 0) \wedge \neg j \rightarrow (t > 0) \\
\wedge \quad & j \rightarrow ( \quad (\mathit{off} \wedge \mathit{on}' \\
& \qquad \wedge T_{\mathcal{I}} \geq -16 \wedge T'_{\mathcal{I}} = T_{\mathcal{I}} \wedge p' = p \\
& \qquad \wedge T_{\mathcal{I}} \leq -16 \wedge T'_{\mathcal{I}} \geq -20) \\
& \quad \vee (\mathit{off} \wedge \mathit{open}' \\
& \qquad \wedge p \geq 0 \wedge T'_{\mathcal{I}} = T_{\mathcal{I}} \wedge p' = 0 \\
& \qquad \wedge T_{\mathcal{I}} \leq -16 \wedge p' \leq \frac{3}{60}) \\
& \quad \vee (\mathit{on} \wedge \mathit{off}' \\
& \qquad \wedge T_{\mathcal{I}} \leq -20 \wedge T'_{\mathcal{I}} = T_{\mathcal{I}} \wedge p' = p \\
& \qquad \wedge T_{\mathcal{I}} \geq -20 \wedge T'_{\mathcal{I}} \leq -16) \\
& \quad \vee (\mathit{on} \wedge \mathit{open}' \\
& \qquad \wedge p \geq 0 \wedge T'_{\mathcal{I}} = T_{\mathcal{I}} \wedge p' = 0 \\
& \qquad \wedge T_{\mathcal{I}} \geq -20 \wedge p' \leq \frac{3}{60}) \\
& \quad \vee (\mathit{open} \wedge \mathit{off}' \\
& \qquad \wedge T_{\mathcal{I}} \leq -16 \wedge T'_{\mathcal{I}} = T_{\mathcal{I}} \wedge p' = -2p \\
& \qquad \wedge p \leq \frac{3}{60} \wedge T'_{\mathcal{I}} \leq -16) \\
& \quad \vee (\mathit{open} \wedge \mathit{on}' \\
& \qquad \wedge T_{\mathcal{I}} \geq -20 \wedge T'_{\mathcal{I}} = T_{\mathcal{I}} \wedge p' = -2p \\
& \qquad \wedge p \leq \frac{3}{60} \wedge T'_{\mathcal{I}} \geq -20)) \\
\wedge \neg j \rightarrow ( \quad & (\mathit{off} \wedge \mathit{off}' \\
& \qquad \wedge T'_{\mathcal{I}} = (T_{\mathcal{I}} - 20)e^{-0.3t} + 20 \wedge p' = p + t \\
& \qquad \wedge T_{\mathcal{I}} \leq 16 \wedge T'_{\mathcal{I}} \leq 16) \\
& \quad \vee (\mathit{on} \wedge \mathit{on}' \\
& \qquad \wedge T'_{\mathcal{I}} = (T_{\mathcal{I}} + \frac{80}{3})e^{-0.3t} - \frac{80}{3} \wedge p' = p + t \\
& \qquad \wedge T_{\mathcal{I}} \geq 20 \wedge T' \geq 20) \\
& \quad \vee (\mathit{open} \wedge \mathit{open}' \\
& \qquad \wedge T'_{\mathcal{I}} = (T_{\mathcal{I}} - 20)e^{-2.2t} + 20 \wedge p' = p + t \\
& \qquad \wedge p \leq \frac{3}{60} \wedge p' \leq \frac{3}{60}))
\end{aligned}
$$

To generate the subformulae $I(\vec{y}^{\,0})$ and $R(\vec{y}^{\,k})$ of $\phi_k$, we apply encoding steps 6 and 7. Being interested in the reachability of states where $T_{\mathcal{I}}$ exceeds $0°C$, we add $T_{\mathcal{I}}^k \geq 0$ to $R(\vec{y}^{\,k})$. Furthermore, we extend $R(\vec{y}\,)$ by $\mathit{off}^k + \mathit{on}^k + \mathit{open}^k = 1$, because the transition predicate $T(\vec{y}\,, \vec{y}\,')$ does not enforce mutual exclusion of discrete control modes for the very last step of an execution. Doing so, we obtain the following formulae:

$$\begin{aligned} I(\vec{y}^{\,0}) := \quad & \mathit{off}^0 \rightarrow (T_{\mathcal{I}}^0 \leq -16 \ \wedge \ p^0 = 0) \\ & \wedge \ \mathit{on}^0 \rightarrow (T_{\mathcal{I}}^0 \geq -20 \ \wedge \ p^0 = 0) \\ & \wedge \ \mathit{open}^0 \rightarrow \mathit{false} \end{aligned}$$

$$\begin{aligned} R(\vec{y}^{\,k}) := \quad & T_{\mathcal{I}}^k \geq 0 \\ & \wedge \ \mathit{off}^k + \mathit{on}^k + \mathit{open}^k = 1 \end{aligned}$$

The resultant system description, consisting of $T(\vec{y}\,, \vec{y}\,')$, $I(\vec{y}^{\,0})$, and $R(\vec{y}^{\,k})$, is very close to the input format processed by the solvers which we present in the following three chapters. When being fed with above formulae, the HySAT-2 solver, which will be introduced in chapter 5, will construct and solve $\phi_k$ for increasing values of $k$, starting with $k = 0$. For $0 \leq k \leq 25$, HySAT-2 diagnoses the formula to be unsatisfiable. For $k \geq 26$, satisfying valuations, corresponding to executions refuting the safety property under investigation, exist.

Figure 2.4 on the next page shows an error trace which was obtained from a solution of $\phi_{27}$ found by HySAT-2. Despite adherence to the prescribed policy for opening and closing the door, the temperature inside the freezer cell evolves from initially $-18°C$ to $0°C$ within less than $1.2$ hours.[4]

Because of the monotonicity of all continuous flows and the linearity of mode invariants and transition guards, the encoding of the freezer example is exact in the sense that satisfying valuations of the BMC formula are in one-to-one correspondance with error traces. Note, however, that in general the formulae generated by the above translation scheme overapproximate the dynamics of the encoded hybrid automaton. This is due to the fact that the formulae only talk about starting points and endpoints of flows and are thus blind to violations of mode invariants and enabledness of transition guards which occur at intermediate time points. Solutions of such formulae may describe spurious traces where a flow of, e.g., parabolic shape starts in a state satisfying

[4] Note that although the evolution of $T_{\mathcal{I}}$ during flows might appear linear in the chart, it is indeed composed of moderately curved exponential function segments.

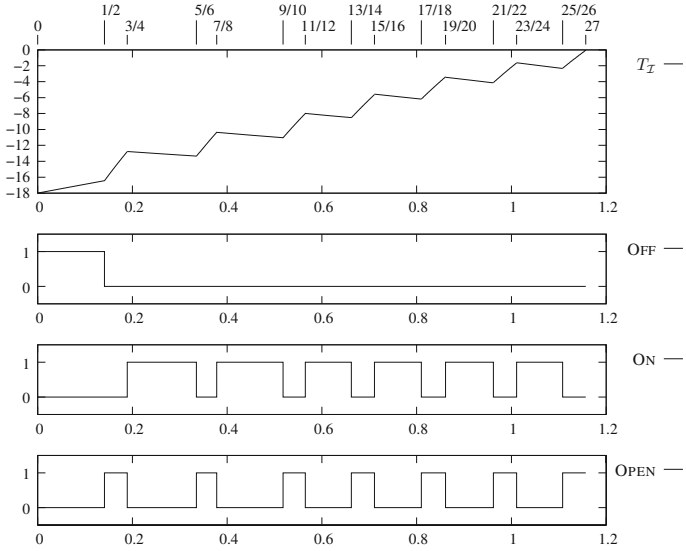


Figure 2.4: Error trace of the freezer automaton.

the current mode invariant and finally returns to such a state, but temporarily leaves the invariant region in between. This can also happen in case of monotonic flows if the mode invariant describes a non-convex region in the state space. Often an encoding can be made exact, e.g. by adding constraints that confine flows to monotonic segments. This requires however a thorough understanding of the system dynamics.

### 2.2.2 Hybridization of Continuous Dynamics

If the differential equations of a hybrid automaton do not permit the computation of closed-form solutions, the above translation scheme is not applicable directly. In this case we can resort to transforming the hybrid automaton into one which safely approximates the original continuous dynamics with simpler dynamics that we can deal with in the encoding. This is also the way to go if the closed-form solutions are too complex to be handled by the verification backend, e.g. nonlinear, while the solver to be used for BMC only supports linear arithmetic.

A method for automatically constructing an approximate system with guaranteed error bound is *hybridization*. See [6] for a thorough introduction into the topic, including a survey of related work.

The basic idea behind hybridization is to relax the flow predicate $p$ of a control mode $v$ using a weaker predicate $p'$, such that $p$ implies $p'$, where the dynamics described by $p'$ is of simpler type than the one described by $p$. The overapproximation thus obtained can be refined by splitting the continuous state space into cells $Q_1, \ldots, Q_l$, and replacing $v$ with new modes $v_1, \ldots, v_l$, where each mode $v_i$ describes $v$'s continuous dynamics within $Q_i$. Each new mode inherits all incoming and outgoing discrete transitions of $v$. Furthermore, discrete transitions are added which allow to pass over between modes representing adjacent cells in the state space. The restriction of flows to smaller regions in the state space allows to strengthen the flow predicates of the individual modes and thereby to tighten the approximation.

By refining the partitioning of the state space, i.e. by reducing the size of the cells and at the same time increasing their number, the approximation can in principle be made arbitrarily tight. However, tighter approximations yield hybrid automata with more control modes, which renders analysis more expensive.

**Example 2.10.** Consider again the hybrid automaton depicted in figure 2.3 on page 23. In order to simplify the dynamics of control mode OPEN, let us assume that for checking the property in question it is sufficient to consider flows where $T_{\mathcal{I}}$ does not leave the interval $[-20, 0]$. We therefore conjoin $-20 \leq T_{\mathcal{I}} \leq 0$ to the mode invariant. Since $(\dot{T}_{\mathcal{I}} = -2.2\ T_{\mathcal{I}} + 44) \wedge (-20 \leq T_{\mathcal{I}} \leq 0)$ implies that the first derivative of $T_{\mathcal{I}}$ is bounded from below by $55$ and bounded from above by $77$, the original flow predicate $\dot{T}_{\mathcal{I}} = -2.2\ T_{\mathcal{I}} + 44$ can be relaxed to $55 \leq \dot{T}_{\mathcal{I}} \leq 77$, yielding a first, yet coarse, overapproximation of the continuous dynamics in mode OPEN.

The relaxed flow predicate is not a differential equation, but a differential inequality which specifies for each initial value $T_{\mathcal{I}}(0)$ a bundle of trajectories. This bundle is confined to a linear envelope which is bounded by the flows where $T_{\mathcal{I}}$ evolves with minimum or maximum slope throughout. Hence, the value of $T_{\mathcal{I}}$ after residing $t$ time units in mode OPEN lies in the interval $[\,T_{\mathcal{I}}(0) + 55t,\ T_{\mathcal{I}}(0) + 77t\,]$. Using this, we can amend the transition predicate $T(\vec{y}, \vec{y}\,')$, given on page 28, in order to match the changes in the automaton. To this end, we replace the last disjunct in the subformula generated by encoding step 5 with

$$
\begin{aligned}
&(\mathit{open} \wedge \mathit{open}' \\
&\wedge\ (T_{\mathcal{I}} + 55t) \leq T'_{\mathcal{I}} \leq (T_{\mathcal{I}} + 77t) \wedge p' = p + t \\
&\wedge\ (-20 \leq T_{\mathcal{I}} \leq 0) \wedge (-20 \leq T'_{\mathcal{I}} \leq 0) \wedge p \leq \frac{3}{60} \wedge p' \leq \frac{3}{60}).
\end{aligned}
$$

In figure 2.5 on page 35, the initial approximation constructed above has been refined by splitting control mode OPEN into four new modes, each of them representing a strip of 10 degrees Celsius in the state space.[5] The mode invariant of each submode has been strengthened accordingly, which in turn enabled the deduction of stronger flow predicates.

By applying hybridization in the same way to the control modes ON and OFF, we can transform the freezer model into a so-called linear hybrid automaton.

**Definition 2.11.** A hybrid automaton $H = (X, V, E, inv, init, jump, flow)$ is a *linear hybrid automaton* (LHA), if the following requirements are met.

- For every control mode $v \in V$, the flow predicate $flow_v$ is a conjunction of linear arithmetic predicates over $\dot{X}$.
- For every control mode $v \in V$, the predicates $inv_v$ and $init_v$ are conjunctions of linear arithmetic predicates over $X$.
- For every discrete transition $e \in E$, the jump condition $jump_e$ is a Boolean combination of linear arithmetic predicates over $X \cup X'$.

Linear hybrid automata are a simple, yet powerful class of hybrid systems. They are complex enough to be relevant for practical applications, and at the same time their continuous dynamics is simple enough to be dealt with efficiently in verification algorithms. Predicative encodings of LHAs are Boolean combinations of linear arithmetic constraints. We address solving of such formulae in chapter 4.

Note that for linear hybrid automata the bounded reachability problem, i.e. the problem whether a hybrid state is reachable in a fixed number of steps, is decidable, which follows from the decidability of linear arithmetics over the reals. The general reachability problem for linear hybrid automata is, however, undecidable [73].

In case of the freezer model only the flow predicates had to be relaxed in order to obtain a LHA. If required, it is certainly possible to also replace the predicates defining the location invariants, initial states, and jump conditions with weaker predicates, e.g. with piecewise linear overapproximations, if the desired target format is a linear hybrid automaton.

So far, we presented hybridization as a transformation which is carried out on the automaton level. Alternatively it is possible to construct an encoding which passes the task of hybridizing flows to the solver.

[5] In order not to clutter the picture, we have drawn the new control modes as submodes of mode OPEN. In- and outgoing transitions of the outer mode apply to *all* submodes.

**Example 2.12.** We extend the encoding of the freezer model by adding, for each step $i$, continuous variables $\underline{T_{\mathcal{I}}}^i$ and $\overline{T_{\mathcal{I}}}^i$. Through $\underline{T_{\mathcal{I}}} \leq T_{\mathcal{I}} \leq \overline{T_{\mathcal{I}}}$ they define a cell in the state space that must not be left during a flow. By adding the constraint

$$\neg j \rightarrow (\overline{T_{\mathcal{I}}} - \underline{T_{\mathcal{I}}} \leq \delta)$$

to the transition predicate $T(\vec{y}, \vec{y}')$ of the freezer model, we ensure that in case of a flow starting at step $i$ of the execution $\underline{T_{\mathcal{I}}}^i$ and $\overline{T_{\mathcal{I}}}^i$ form an interval of width less or equal than $\delta$, where $\delta > 0$ is a parameter which controls the tightness of the overapproximation. Exploiting that for $T_{\mathcal{I}} \in [\underline{T_{\mathcal{I}}}, \overline{T_{\mathcal{I}}}]$ the first derivative of $T_{\mathcal{I}}$ is bounded by $-2.2\ \overline{T_{\mathcal{I}}} + 44$ from below and by $-2.2\ \underline{T_{\mathcal{I}}} + 44$ from above, we can encode the flow dynamics of mode OPEN by

$$\begin{aligned}
&(\mathit{open} \wedge \mathit{open}' \\
&\wedge\ (T_{\mathcal{I}} + (-2.2\ \overline{T_{\mathcal{I}}} + 44) \cdot t)\ \leq\ T'_{\mathcal{I}}\ \leq\ (T_{\mathcal{I}} + (-2.2\ \underline{T_{\mathcal{I}}} + 44) \cdot t)\ \wedge\ p' = p + t \\
&\wedge\ (\underline{T_{\mathcal{I}}} \leq T_{\mathcal{I}} \leq \overline{T_{\mathcal{I}}})\ \wedge\ (\underline{T_{\mathcal{I}}} \leq T'_{\mathcal{I}} \leq \overline{T_{\mathcal{I}}})\ \wedge\ p \leq \frac{3}{60}\ \wedge\ p' \leq \frac{3}{60})
\end{aligned}$$

which allows the solver to determine a suitable partition of the state space into cells at runtime.

As opposed to hybridization performed on the automaton level, this approach yields smaller formulae because it avoids an explicit construction and encoding of the submodes arising from mode-splitting. The constraints occuring in the formula are, however, more complex (polynomial instead of linear in the above example), since they involve the computation of upper and lower bounds on the derivatives.

### 2.2.3 Encoding Flows Using Taylor Expansions

A systematic way to determine relaxations of flow curves as needed for hybridization, is to use Taylor polynomials together with an interval bounded remainder term which encloses the approximation error.

**Example 2.13.** To demonstrate the use of Taylor expansions, we consider again the differential equation $\frac{dT_{\mathcal{I}}}{dt} = -2.2\ T_{\mathcal{I}} + 44$, i.e. the first derivative of the unknown function describing the flow in control mode OPEN. By differentiation, we can easily compute higher order derivatives, e.g.

$$\frac{d^2T_{\mathcal{I}}}{dt^2} = -2.2 \cdot \frac{dT_{\mathcal{I}}}{dt} = 4.84 \cdot T_{\mathcal{I}} - 96.8.$$

Using these derivatives, we can can approximate the solution function of the differential equation by the $n$-th order Taylor polynomial

$$p_n(t_0 + t) = \sum_{k=0}^{n} \frac{1}{k!} \cdot \frac{d^k T_{\mathcal{I}}}{dt^k}(t_0) \cdot t^k,$$

where $\frac{d^0 T_{\mathcal{I}}}{dt^0}(t_0) = T_{\mathcal{I}}(t_0)$ is the initial value of the flow. The approximation error increases with increasing values of $t$. Taylor's theorem provides that there exists a $\xi$ between $t_0$ and $t_0 + t$ such that the error $T_{\mathcal{I}}(t_0 + t) - p_n(t_0 + t)$ is given by

$$r_n(t) = \frac{1}{(n+1)!} \cdot \frac{d^{n+1} T_{\mathcal{I}}}{dt^{n+1}}(\xi) \cdot t^{n+1}.$$

For $n = 1$, e.g., we obtain the Taylor polynomial $p_1(t_0+t) = T_{\mathcal{I}}(t_0)+(-2.2\ T_{\mathcal{I}}(t_0)+44) \cdot t$. The corresponding remainder term is $r_1(t) = \frac{1}{2} \cdot (4.84\ T_{\mathcal{I}}(\xi) - 96.8) \cdot t^2$ for some $\xi \in [t_0, t_0 + t]$.

If we constrain the flow, as in the previous example, to a cell defined by $\underline{T_{\mathcal{I}}} \leq T_{\mathcal{I}} \leq \overline{T_{\mathcal{I}}}$, then the remainder is bounded by $\underline{r_1}(t) = \frac{1}{2} \cdot (4.84\ \underline{T_{\mathcal{I}}} - 96.8) \cdot t^2$ from below and by $\overline{r_1}(t) = \frac{1}{2} \cdot (4.84\ \overline{T_{\mathcal{I}}} - 96.8) \cdot t^2$ from above. Hence, the endpoint $T_{\mathcal{I}}(t_0 + t)$ of the flow after $t$ time units is enclosed in the interval $[\, p_1(t_0+t)+\underline{r_1}(t),\ p_1(t_0+t)+\overline{r_1}(t)\,]$. In the BMC formula we can therefore relate starting point and endpoint of the flow through

$$\begin{aligned} T'_{\mathcal{I}} \ &\geq \ \left(T_{\mathcal{I}} + (-2.2\ T_{\mathcal{I}} + 44) \cdot t + \frac{1}{2} \cdot (4.84\ \underline{T_{\mathcal{I}}} - 96.8) \cdot t^2\right) \\ \wedge\ T'_{\mathcal{I}} \ &\leq \ \left(T_{\mathcal{I}} + (-2.2\ T_{\mathcal{I}} + 44) \cdot t + \frac{1}{2} \cdot (4.84\ \overline{T_{\mathcal{I}}} - 96.8) \cdot t^2\right). \end{aligned}$$

Note, that the relaxed flow constraints used in section 2.2.2 are in fact Taylor expansions of order zero. Higher order Taylor expansions, like the one developed above, define nonlinear boundary curves which enclose the flow more tightly for a longer duration. As the order of the Taylor polynomial increases, the fit increases, too. The higher accuracy of the approximation comes however (again) at the price of more complex constraints to be solved.

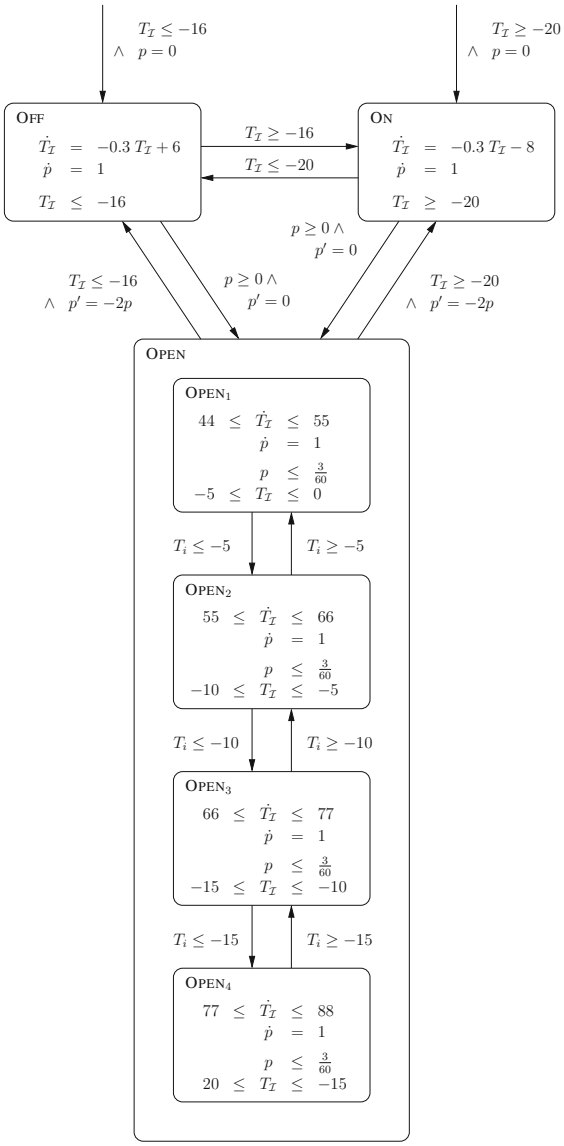


Figure 2.5: Hybridization of the dynamics in control mode OPEN.

# 3 Extending DPLL for Pseudo-Boolean Constraints

As many verification and design automation problems concerning finite state hardware and software systems can be expressed as satisfiability problems for propositional logics, there is a growing demand for efficient satisfiability checkers for such logics. Application domains include combinational and sequential equivalence checking for circuits, test pattern generation, and bounded model checking . More recently, SAT solvers have become essential components in SMT solvers, where they are coupled with solvers for conjunctions of constraints over some background theory (e.g. the theory of linear arithmetic over the reals) in order to solve arbitrary Boolean combinations of such constraints. SAT solvers thus take a key role in technologies which help to master the complexity of ever larger circuits and ever more refined embedded software, which has sparked much research on enhancing their capabilities.

Concerning performance, the most dramatic improvements have been achieved on SAT solvers for conjunctive normal forms (CNFs) that implement variants of the DPLL procedure. As in the classical DPLL procedure, the main algorithmic ingredients of these solvers are unit propagation and backtrack search. These have, however, been enhanced by heuristics for finding a suitable traversal sequence through the backtrack search tree, as well as by refined algorithms and data structures for pruning the search tree and for accelerating unit propagation. Considerable search tree pruning has been achieved through non-chronological backtracking [68, 112, 123, 91] and conflict-driven learning [112, 123, 91], usually combined with random or periodic restarts [9, 91]. Unit propagation is sped up through dedicated data structures [124, 123] and through lazy clause evaluation [91], which delays re-evaluation of the truth value of any clause that is definitely non-unit.

While these techniques actually yield a dramatic speedup in practice, now tackling instances with hundreds of thousands of propositions, which would have been completely impractical a decade ago, they still reach their limits for state-of-the-art verification problems derived from high-level design models (e.g., STATEMATE models [70]) of embedded software. Such models easily yield CNFs with millions of proposi-

C. Herde, *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure*, DOI 10.1007/978-3-8348-9949-1_3,

tional variables under bounded model checking. While some of this complexity is inherent to the verification task, another part can be attributed to the low expressiveness of conjunctive normal forms. Apart from improving CNF solvers performance-wise, another line of research is therefore aiming at extending such procedures to handle other, more expressive types of constraints.

In this chapter we will study how to extend a DPLL-based SAT solver to efficiently solve conjunctions of pseudo-Boolean constraints. Compared to clauses of a CNF, pseudo-Boolean constraints, i.e. linear inequalities on Boolean variables, allow a significantly more compact description of many discrete problems. Pseudo-Boolean constraints arise naturally in many application domains and have been used to efficiently encode problems from electronic design automation, formal verification, and operations research. Pseudo-Boolean problems have usually been handled by integer linear programming (ILP) solvers. The drawback is that the latter do not take into account the Boolean nature of the problem and thus cannot apply the specialized methods exploited in SAT checkers. In principle, pseudo-Boolean constraints can be encoded as pure CNF formulae and be solved by a conventional SAT solver. However a naive conversion of a pseudo-Boolean constraint into CNF can require an exponential number of clauses, thus preventing the solver from effectively processing the search space. Actually, this increase in problem size can be reduced from exponential to linear by introducing auxiliary variables, however leading to a worst-case exponential blow-up in the size of the backtrack search tree.

It has been observed by Barth [13] that the DPLL procedure can easily be modified to handle pseudo-Boolean constraints *directly*, although this clearly introduces some overhead in the satisfiability engine caused by the more complex data structures it uses for reasoning. Whittemore, Kim, and Sakallah tried to follow up on the advances in the algorithmics of CNF-SAT solvers by adapting GRASP's conflict analysis and learning to zero-one linear constraint systems [120], and Aloul, Ramani, Markov, and Sakallah ported CHAFF's lazy clause evaluation to this setting [3]. Yet, they simply mimicked CHAFF's lazy evaluation scheme such that their type of lazy clause evaluation is confined to the pure CNF part of the problem, i.e. applies only to those clauses that are disjunctive. As all other clauses are evaluated eagerly, and as clause re-evaluation is known to account for the major part of the runtime of a DPLL-based SAT solver [91], this is far from optimal. In this chapter, we will show that it is possible and effective to generalize lazy clause evaluation to arbitrary linear constraints under a zero-one interpretation. Closely related techniques have been independently devised and implemented by Chai and Kuehlmann [32].

We will provide a brief introduction to pseudo-Boolean constraints in Section 3.1 and to the state of the art in CNF-SAT in Section 3.2. Section 3.3 explains the algorithms and data structures underlying our SAT solver for pseudo-Boolean constraints, which incorporates generalized DPLL, conflict-driven learning, and lazy clause evaluation for pseudo-Boolean constraints. Section 3.4 provides benchmark results, followed by a conclusion in section 3.5.

## 3.1 The Logics

We aim at solving conjunctions of constraints which are linear inequalities over Boolean variables, so-called pseudo-Boolean constraints. More precisely, a *linear pseudo-Boolean constraint* is of the form

$$a_1x_1 + a_2x_2 + \ldots a_nx_n \geq k \ ,$$

where the $x_i$ are *literals*, i.e. positive or negated *propositional variables*, the $a_i$ are natural numbers, called the *weights* of the individual literals, and $k \in \mathbb{N}$ is the *threshold*. A special form are *cardinality constraints*, where all weights are 1.

We refer to conjunctions of pseudo-Boolean constraints as *zero-one linear constraint systems (ZOLCS)* or *linear pseudo-Boolean constraint systems* [13]. Let $BV$ be a countable set of Boolean variables. Then the syntax of zero-one linear constraint systems is as follows.

$$\begin{aligned}
\mathit{formula} &::= \{\mathit{linear_PB_constraint} \wedge\}^* \mathit{linear_PB_constraint} \\
\mathit{linear_PB_constraint} &::= \mathit{linear_term} \geq \mathit{threshold} \\
\mathit{linear_term} &::= \{\mathit{weight\ literal} +\}^* \mathit{weight\ literal} \\
\mathit{weight} &::\in \mathbb{N} \\
\mathit{literal} &::= \mathit{boolean_var} \mid \overline{\mathit{boolean_var}} \\
\mathit{boolean_var} &::\in BV \\
\mathit{threshold} &::\in \mathbb{N}
\end{aligned}$$

Pseudo-Boolean constraints and conjunctions thereof are interpreted over Boolean *valuations* $\sigma : BV \xrightarrow{\text{total}} \mathbb{B}$ of the propositional variables. We say that $\sigma$ *satisfies* a zero-one linear constraint system $\phi$, denoted $\sigma \models \phi$, iff $\sigma$ satisfies all pseudo-Boolean constraints in $\phi$. A Boolean valuation $\sigma$ satisfies the pseudo-Boolean constraint $a_1x_1+$

$a_2x_2 + \ldots a_nx_n \geq k$ iff $a_1\chi_\sigma(x_1) + a_2\chi_\sigma(x_2) + \ldots a_n\chi_\sigma(x_n) \geq k$, where

$$\chi_\sigma(x) = \begin{cases} 0 & \text{if } x \in BV \text{ and } \sigma(x) = \texttt{false}, \\ 1 & \text{if } x \in BV \text{ and } \sigma(x) = \texttt{true}, \\ 1 - \chi_\sigma(y) & \text{if } x \equiv \overline{y} \text{ for some } y \in BV, \end{cases}$$

i.e. if the left-hand side of the inequality evaluates to a value exceeding the threshold when the truth values `false` and `true` are identified with $0$ and $1$, respectively.

**Example 3.1.** The pseudo-Boolean constraint $5a + 3\overline{b} + 3c + 1d \geq 7$, where $\overline{b}$ denotes the negation of $b$, is satisfied by the valuation $a \mapsto \texttt{false}, b \mapsto \texttt{false}, c \mapsto \texttt{true}, d \mapsto \texttt{true}$, yet is not satisfied by the valuation $a \mapsto \texttt{true}, b \mapsto \texttt{true}, c \mapsto \texttt{false}, d \mapsto \texttt{true}$.

Pseudo-Boolean constraints can represent a wide class of Boolean functions, e.g. $1a + 1b + 1\overline{c} + 1d \geq 1$ is equivalent to $a \vee b \vee \overline{c} \vee d$, $1a + 1b + 1\overline{c} + 1d \geq 4$ is equivalent to $a \wedge b \wedge \overline{c} \wedge d$, and $1a + 1b + 3\overline{c} + 1d \geq 3$ is equivalent to $c \rightarrow (a \wedge b \wedge d)$. In fact, zero-one linear constraint systems can be exponentially more concise than CNFs. A CNF expressing that at least $n$ out of $k$ variables should be true requires $\binom{n}{k}$ clauses of length $n$ each, i.e. is of size $O\left(\binom{n}{k} \cdot n\right)$, whereas the corresponding ZOLCS has size linear in $k$ and logarithmic in $n$.

When solving pseudo-Boolean satisfiability problems with Davis-Putnam-like procedures, we will build valuations incrementally such that we have to reason about *partial valuations* $\rho : BV \xrightarrow{\text{part.}} \mathbb{B}$ of the propositional variables. We say that a variable $v \in BV$ is *unassigned in* $\rho$ iff $v \notin \mathrm{dom}(\rho)$. A partial valuation $\rho$ is called *consistent for a formula* $\phi$ iff there exists a total extension $\sigma : BV \xrightarrow{\text{total}} \mathbb{B}$ of $\rho$ that satisfies $\phi$. Otherwise, we call $\rho$ *inconsistent for* $\phi$. Furthermore, a partial valuation $\rho$ is said to *satisfy* $\phi$ iff all its total extensions satisfy $\phi$. As this definition of satisfaction agrees with the previous one on total valuations, we will use the same notation $\rho \models \phi$ for satisfaction by partial and by total valuations.

## 3.2 State of the Art in SAT Solving

Before reporting on the methods employed in modern SAT solvers, we recall some basic facts and notions concerning Boolean formulae.

*Notation*

Every Boolean function $f : \mathbb{B}^k \to \mathbb{B}$ can be expressed as a Boolean formula built from $k$ propositional variables and the Boolean operations *conjunction*, *disjunction*, and *negation*, represented by the symbols $\wedge$, $\vee$, and $\neg$, respectively. As usual, the implication $a \to b$, which can also be written as $b \leftarrow a$, is defined to mean $(\neg a \vee b)$, and the equivalence operation $a \leftrightarrow b$ means $(a \to b) \wedge (a \leftarrow b)$. As we identify the truth values with the natural numbers 0 and 1, we occasionally write $a \leq b$ instead of $a \to b$, similarly $a \geq b$ instead of $a \leftarrow b$, and $a = b$ instead of $a \leftrightarrow b$.

*Satisfiability and Falsifiability*

A Boolean formula is *satisfiable* if a valuation exists which makes the formula evaluate to true. A formula which is satisfied by any valuation, is called *valid* or *tautology*. Two formulae $f$ and $g$ are said to be logically *equivalent*, denoted by $f \equiv g$, if $f \leftrightarrow g$ is a tautology. The dual concept of satisfiability is falsifiability. A Boolean formula is *falsifiable* if a valuation exists which makes the formula evaluate to false. A formula which is falsified by any valuation, is called *unsatisfiable* or *contradiction*.

*Normal Forms*

A Boolean formula is said to be in *conjunctive normal form* (CNF) or *clausal form* if it is a conjunction of clauses, where a clause is a disjunction of literals. As before, a literal is a propositional variable or its negation. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of terms, the latter being conjunctions of literals. Any Boolean formula can be transformed into an equivalent formula in CNF or DNF, respectively.

A formula is in *negation normal form* (NNF) if negations only occur within literals, and, apart from that, $\wedge$ and $\vee$ are the only Boolean connectives, i.e. the formula does not contain implications or equivalences. Obviously, CNF and DNF are special cases of the negation normal form.

*Complexity*

Given a Boolean formula $f$, we refer to SAT as the problem to decide whether $f$ is satisfiable, to FAL as the problem to decide whether $f$ is falsifiable, to VALID as the problem to decide whether $f$ is a tautology, and to UNSAT as the problem to decide whether $f$ is a contradiction. The SAT problem for Boolean formulae was the first problem known to be NP-complete. The complementary problem UNSAT

| | SAT | VALID | FAL | UNSAT |
|---|---|---|---|---|
| arbitrary form | NP | coNP | NP | coNP |
| conjunctive normal form | NP | linear | linear | coNP |
| disjunctive normal form | linear | coNP | NP | linear |

Table 3.1: Complexity of decision problems.

is coNP-complete. We can determine if a formula is falsifiable by checking whether its negation is satisfiable. Therefore FAL is, like SAT, NP-complete. Likewise, we can determine if a formula is valid by checking whether its negation is unsatisfiable. Therefore VALID is, like UNSAT, coNP-complete. If the formulae are, however, in conjunctive or disjunctive normal form, then the results are different. Validity (and therefore falsifiability) of a CNF can be checked in time linear in the size of the formula, because a CNF is a tautology iff each clause is a tautology, i.e. iff each clause has a subclause of form $(b \vee \neg b)$. Unsatisfiability (and therefore satisfiability) of a DNF is linear, because a DNF is a contradiction iff each term is a contradiction, i.e. iff each term has a subterm of form $(b \wedge \neg b)$. Table 3.1 summarizes the quoted complexity results.

Given that any Boolean formula can be translated into an equivalent formula in DNF and CNF, it may be surprising that checking DNF-satisfiability and CNF-unsatisfiability can be done in linear time, while the corresponding problems for general Boolean formulae are NP- and coNP-complete, respectively. The crucial point is that presumably, unless P = NP, no equivalence-preserving translation with polynomial time complexity exists.

### 3.2.1 Conversion into CNF

At present, the most powerful SAT solvers are based on the DPLL procedure which requires its input to be in conjunctive normal form. An efficient translation of a Boolean formula into clausal form is therefore essential for satisfiability checking.

*Naive Translation*

The standard conversion to CNF, which can be found in most textbooks on propositional logic, first transforms the input formula into NNF by using De Morgan's laws to push all negations down the expression tree until they only appear immediately above variables, i.e. within literals. Then disjunctions are pushed towards the literals

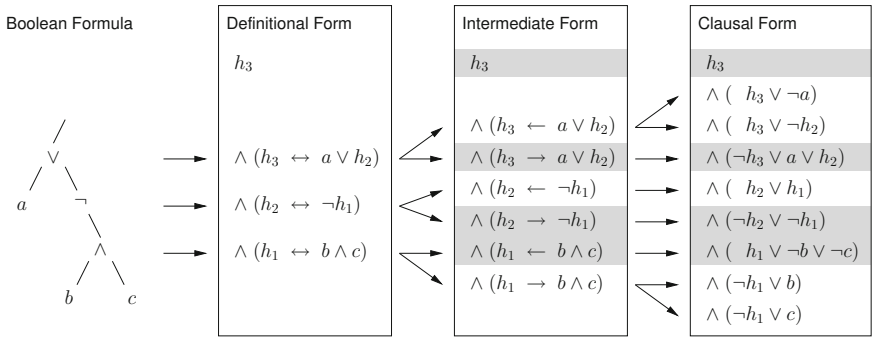


Figure 3.1: Definitional translation.

by application of the distributivity laws of Boolean logic, resulting in a CNF which is logically equivalent to the input formula, however potentially exponentially larger.

*Definitional Translation*

To avoid the exponential blow-up in formula size, the *definitional translation* drops the goal of converting the input formula into a logically equivalent formula in clausal form, but merely aims at constructing a CNF which is satisfiable if and only if the input formula is satisfiable. It is therefore said to be *satisfiability-preserving* (rather than *equivalence-preserving*). The definitional translation is often referred to as *Tseitin translation*, because an early description was given by G. S. Tseitin [118].

The idea behind the definitional translation is that the satisfiability of a formula $f$ is not affected by replacing a subformula $g$ in $f$ with a fresh propositional variable $h_g$ and conjoining the constraint $(h_g \leftrightarrow g)$, referred to as *definition*, to the resulting formula $f[h_g/g]$. The formulae $f$ and $f[h_g/g] \wedge (h_g \leftrightarrow g)$ are *equisatisfiable*, meaning that each solution of the former corresponds to a solution of the latter and vice versa. Moreover, satisfying valuations coincide on the variables which occur in both formulae. The new variable $h_g$, which acts as abbreviation for $g$ within $f$, is called *Tseitin variable* or *Tseitin label* of $g$.

By applying the above transformation-rule in a bottom-up fashion to all inner nodes of the expression tree, the input formula is translated into a conjunction of the Tseitin variable representing the top-node of the formula, and definitions whose number is linear in the size of $f$. This representation of $f$ — we refer to it as *defi-*

*nitional form* — can be easily translated into CNF by rewriting each definition into a small set of clauses.

It has been observed by Plaisted and Greenbaum [102], that the definitional translation can be made more succinct by taking the polarity of a subformula into account. A subformula $g$ is said to occur positively (or with positive polarity) in $f$ if it occurs below an even number of negation signs. Otherwise, $g$ is said to occur negatively (or with negative polarity) in $f$. Depending on the polarity of a subformula, only one of the two implications involved in the definition is needed to constrain the corresponding Tseitin variable: If $g$ occurs positively in $f$, then $f$ is rewritten to $f[h_g/g] \wedge (h_g \rightarrow g)$, and to $f[h_g/g] \wedge (h_g \leftarrow g)$ if $g$ has negative polarity in $f$. Compared to the unoptimized definitional translation, a translation with polarity optimization saves half of the CNF clauses on average.

**Example 3.2.** Figure 3.1 on the previous page illustrates the definitional translation of $\phi = a \vee \neg(b \wedge c)$ into CNF. Using fresh propositional variables $h_1$, $h_2$, and $h_3$, $\phi$ is first converted into definitional form. Clausal form is obtained by splitting each definition into two implications, and then rewriting each implication into (at most two) clauses. Only the highlighted implications and clauses have to be kept according to polarity optimization.

We use the definitional translation, enhanced with polarity optimization, in all solvers described in this thesis. In addition, we apply several simple optimizations, among them the following.

- For negation nodes in the expression tree, representing subformulae of shape $\neg g$, no separate Tseitin labels are introduced. Instead, the negation of the Tseitin variable for $g$ is used as label for $\neg g$.

- If there are multiple occurences of the same subformula, then all occurrences share the same Tseitin variable. In particular, we ensure that the same Tseitin variable is used for positive and negative occurences of a subformula, as well as for subformulae which have same polarity but are, like $(b \wedge c)$ and $(\neg b \vee \neg c)$, negations of each other.

- In order to optimize the potential for sharing of Tseitin labels, we apply (previous to the CNF conversion) various normalization rules to the input formula which aim at transforming logically equivalent subformulae into syntactically equivalent ones.

*Mixed Translation Schemes*

It is not difficult to find cases where the definitional translation performs worse than the standard conversion. The formula from the previous example, e.g., can be expressed as a single clause, namely $(a \vee \neg b \vee \neg c)$, whereas the definitional translation with polarity optimization generates a CNF consisting of four clauses. Boy de la Tour [44], Nonnengart and Weidenbach [96], and Jackson and Sheridan [76] therefore proposed mixed translation schemes, which aim at introducing a definition for a subformula only if doing so does not increase the number of clauses generated when compared to the standard translation, and resort to the latter otherwise.

Experimental results presented by Jackson and Sheridan [77] demonstrate that their translation scheme consistently generates smaller CNFs than a pure definitional approach. Unsurprisingly, the results also indicate that a smaller CNF is not necessarily solved faster — an observation which is in accordance with the finding in [103]. In our solvers, we therefore opted for using a plain definitional translation.

### 3.2.2 SAT Checkers for CNFs

Because of their relevance for practical applications, and probably also since they might provide a key to the fundamental question whether P = NP, algorithms for SAT solving have been investigated for almost 50 years by now. Accordingly large is the amount of literature covering the field. In this section, we can only give a brief survey on the most important techniques employed in modern SAT solvers. In particular, we focus on complete solvers based on the Davis-Putnam-Loveland-Logemann (DPLL) procedure, which are the most efficient SAT solvers to date. Aiming at using SAT solvers in formal verification, we ignore incomplete solvers, which are not guaranteed to find existing satisfying valuation, and are therefore not suitable for applications like bounded model checking. For more comprehensive accounts on SAT solving we refer the reader to the survey articles by Zhang and Malik [126], and Gu et al. [69].

*DPLL Procedure*

Most modern implementations of complete satisfiability-search procedures are enhancements of the DPLL recursive search procedure, which is given in pseudo-code in Listing 3.1 on the following page. Given a CNF $\phi$ and a partial valuation $\rho$, which is empty at the start, the DPLL procedure incrementally extends $\rho$ until either $\rho \models \phi$ holds or $\rho$ turns out to be inconsistent for $\phi$, in which case another extension is tried through backtracking. Extensions are constructed by either logical deduction based

```
1   DPLL(φ, ρ)
2       ρ := DEDUCE(φ, ρ)
3       if φ contains conflicting clause then return
4       if no free variables left then print ρ; stop
5       b := SELECT_UNASSIGNED_VARIABLE(φ)
6       v := ONE_OF {false, true}
7       DPLL(φ, ρ ∪ {b ↦ v})
8       DPLL(φ, ρ ∪ {b ↦ ¬v})
9
10  DEDUCE(φ, ρ)
11      while unit-clause (x) exists in φ
12          if x ≡ b for some b ∈ V then ρ := ρ ∪ {b ↦ true}
13          if x ≡ b̄ for some b ∈ V then ρ := ρ ∪ {b ↦ false}
14      return ρ
```

Listing 3.1: DPLL procedure in pseudo-code.

on *unit propagation* or by so-called *decisions*, which entail selecting an unassigned variable and 'blindly' assigning a truth-value to it. The DPLL procedure alternates between these two extension strategies, thereby using unit propagation whenever possible.

Unit propagation, as implemented by DEDUCE, checks for occurrence of so-called *unit clauses* and extends the current partial valuation by their implications. A clause is said to be *unit* iff all its literals, except for exactly one unassigned literal, are set to false. To satisfy the CNF $\phi$, the unassigned literal has to be assigned true. That assignment is thus said to be *propagated* by the unit clause. Propagations may make other clauses unit and thus entail further propagations. Unit propagation is iterated until no further unit clauses exist. All assignments triggered by the same decision are said to belong to the same *decision level*.

Deduction may yield a *conflicting clause* which has all its literals assigned false, indicating a dead end in search. Backtracking then backs up to the most recently assigned decision variable which has not yet been tried both ways, thereby undoes all assignments made since assigning that variable, flips its truth-value, and resumes the search. If no such decision variable exists then $\phi$ is unsatisfiable. If, on the other hand,

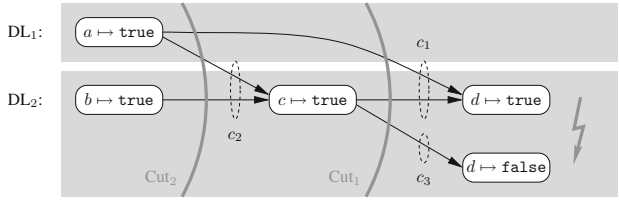


Figure 3.2: Conflict analysis.

all variables of $\phi$ can be assigned without a conflict, a satisfying valuation has been found.

Being based on this recursive search procedure, actual implementations refine the above scheme by using different strategies for selecting the variable and the truth-value to be used at a decision step. See [111] for a detailed review of the impact of different decision strategies on the solving process. Furthermore, state-of-the-art SAT solvers enhance the basic procedure through various algorithmic modifications.

*Conflict Analysis and Conflict-Driven Learning*

Like all pure backtracking algorithms, the DPLL procedure suffers from thrashing, i.e. repeated failure due to the same reason. To overcome this problem, sufficiently general *reasons* for conflicts encountered have to be deduced and stored for future guidance of the search procedure. The standard scheme traces the reason back to a small (ideally minimal) number of assignments that triggered the particular conflict, and stores this reason by adding the negation of that assignment as a clause — termed *conflict clause* — to the clause database. Such reasons, also called *explanation* in this context, can be inferred from cuts in the implication graph. The *implication graph* is a directed graph recording the causal relationship between individual variable assignments performed. It is obtained through relating, for each propagation performed, the propagated literal to the previously assigned literals in the corresponding unit clause. A cut in this graph constitutes a reason for all assignments occurring later in the graph. A more general reason, i.e. an explanation mentioning fewer literals, for an individual assignment can be derived by taking just those elements of a cut that actually have a path to that particular assignment. Using this technique, a reason for the assignments constituting a conflict can be inferred. The reason is, however, not unique, as different cuts can be used. The interested reader is referred to Zhang et al. [125], who examine several heuristics for deriving one or more conflict clauses from the implication graph.

**Example 3.3.** Consider a CNF with clauses $c_1 = (\neg a \vee \neg c \vee d)$, $c_2 = (\neg a \vee \neg b \vee c)$, and $c_3 = (\neg c \vee \neg d)$. Figure 3.2 on the previous page shows the implication graph resulting from the decisions $a \mapsto \texttt{true}$, then $b \mapsto \texttt{true}$. Since $a \mapsto \texttt{true}$ does not trigger any propagations, decision level $\mathrm{DL}_1$ contains only the decision itself. After assigning $b \mapsto \texttt{true}$ on decision level $\mathrm{DL}_2$, clause $c_2$ becomes unit and propagates $c \mapsto \texttt{true}$. Unity of $c_2$ is caused by the valuations $a \mapsto \texttt{true}$ and $b \mapsto \texttt{true}$, which are therefore linked in the implication graph to the propagated assignment $c \mapsto \texttt{true}$. The latter, in turn, entails unity of clause $c_3$ and, together with $a \mapsto \texttt{true}$, clause $c_1$, which propagate the conflicting assignments $d \mapsto \texttt{false}$ and $d \mapsto \texttt{true}$, respectively.

Reasons for the conflict are given by $\mathrm{Cut}_1$ and $\mathrm{Cut}_2$. A cut generates a partition of the graph into the *conflict side*, comprising at least the conflicting assignments, and the *reason side* comprising at least all decisions. All vertices on the reason side which have an outgoing edge crossing the cut, together define a partial valuation which is a reason for the conflict. For example, $\mathrm{Cut}_1$ yields the valuation $\{a \mapsto \texttt{true}, c \mapsto \texttt{true}\}$, which can be excluded from future search by learning the conflict clause $(\neg a \vee \neg c)$. Another conflict clause can be derived from $\mathrm{Cut}_2$, namely $(\neg a \vee \neg b)$.

*Non-Chronological Backtracking*

In addition to pruning the search space, the learned conflict clauses can also be used to accelerate backtracking: instead of just backtracking to the most recent assignment of a decision variable which has not been flipped, the algorithm may directly back up to the maximum decision level on which the conflict clause still has at least one unassigned literal. Doing so can save flipping numerous decisions which do not affect the conflict at all. Because of this non-sequential way of backing up through the levels of the search tree, this technique is referred to as *non-chronological backtracking* or *backjumping*.

*Restarts*

Another enhancement which can be applied in presence of conflict-driven learning is *(random) restarts*. Restarts abort an ongoing search, discard the partial valuation which has been constructed so far, and resume solving from an empty valuation while retaining the learned conflict clauses. This technique is intended to prevent the search from getting lost in non-relevant regions of the search space which might have been entered by early decisions. The restarted solver will not simply repeat its previous run due to the learned conflict clauses, which guide the solver to different parts of the search space. Note, however, that some care has to be taken to preserve completeness of the search process [85], e.g. by limiting the number of restarts.

*Lazy Clause Evaluation*

Concerning performance of DPLL-like procedures, probably the most important improvement in recent years can be attributed to a novel implementation of unit propagation, introduced with the CHAFF SAT solver [91]. Unit propagation in general accounts for the major fraction of the solver runtime.

Previous implementations of unit propagation identified unit clauses by visiting, after each assignment, *all* clauses containing the literal falsified by that assignment, as such clauses might have become unit. The key idea of the enhanced algorithm is to watch only two literals in each clause, and not to visit the clause when any other literal is assigned. This is sound, since the two watched literals provide evidence for the non-unitness of the clause. If an assignment, however, sets a watched literal to false, then this triggers a visit of the respective clause to evaluate its state. The algorithm then tries to replace the watched literal that has been assigned false with another unassigned or true literal occurring in the clause. If it succeeds then it has constructed a new witness for non-unitness of the clause. Otherwise, the clause has become unit and the watched literal which is yet unassigned is the one to be propagated. This technique, often called *lazy clause evaluation*, has been shown to achieve significant performance gains, especially on hard SAT instances [91].

## 3.3 Optimizing DPLL-Based Pseudo-Boolean Solvers

We will now generalize DPLL and its recent enhancements, in particular lazy clause evaluation, to zero-one linear constraint systems. To simplify the exposition we assume in the following that each pseudo-Boolean constraint is rewritten after each assignment according to the following simplification rules:

- A literal which has been set to false is removed from the left-hand side of the constraint as it cannot contribute to its satisfaction any longer. For example, after assigning $b \mapsto \texttt{true}$, the constraint $1a+1\overline{b}+1\overline{c} \geq 2$ becomes $1a+1\overline{c} \geq 2$.

- A literal which has become true is also removed from the constraint, however with its weight subtracted from the threshold of that constraint. For example, after assigning $b \mapsto \texttt{false}$ the constraint $1a+1\overline{b}+1\overline{c} \geq 2$ becomes $1a+1\overline{c} \geq 1$.

Consequently, all literals appearing in constraints are yet unassigned, and indices range over unassigned literals only.

```
1   DEDUCE(φ, ρ)
2     while propagating constraint C exists in φ
3       for each literal x propagated by C
4         if x ≡ b for some b ∈ V then ρ := ρ ∪ {b ↦ true}
5         if x ≡ b̄ for some b ∈ V then ρ := ρ ∪ {b ↦ false}
6     return ρ
```

Listing 3.2: Deduce procedure for ZOLCs.

### 3.3.1 DPLL for Pseudo-Boolean Constraints

Since the seminal work of Barth [13] it is well-known that the basic DPLL procedure can easily be generalized to zero-one linear constraint systems through modification of the deduction procedure.

As before, the task of the deduction routine is to detect propagating constraints and to perform the corresponding assignments. A pseudo-Boolean constraint $\sum a_i x_i \geq k$ propagates a literal $x_j$ iff setting this literal to false would make the constraint unsatisfiable, i.e. iff $(\sum a_i) - a_j < k$.

In contrast to a CNF clause, a pseudo-Boolean constraint can propagate several literals simultaneously. Furthermore, a pseudo-Boolean constraint is not necessarily satisfied after propagation.

**Example 3.4.** When assigning $a$ with `false`, the constraint

$$5a + 3\bar{b} + 3c + 1d + 1e \geq 7$$

propagates $\bar{b}$ and $c$. However, the assignments $b :=$ `false` and $c :=$ `true` do not satisfy the constraint, but merely reduce it to

$$1d + 1e \geq 1.$$

We refer to a pseudo-Boolean constraint which propagates at least one literal as a *propagating constraint*. With this notion, corresponding to the notion of a unit clause in CNF-SAT, we can formulate a generalized DEDUCE procedure for zero-one linear constraint systems as given in Listing 3.2.

Note that a propagating constraint propagates at least the literal with the largest weight appearing in that constraint. If this is not unique because several literals with the same largest weight exist, then all those literals are propagated.

This can be demonstrated by the following argument: Let $x_p$ be the (not necessarily unique) literal with the largest weight appearing in a constraint $\sum a_i x_i \geq k$. Suppose that some literal $x_q$ with $q \neq p$ is propagated by that constraint. Then $(\sum a_i) - a_q < k$ holds due to the propagation rule given above. However, since $c_p \geq c_q$, also $(\sum a_i) - a_p < k$ holds, i.e. $x_p$ is propagated, too.

### 3.3.2 Generalization of Lazy Clause Evaluation

While the generalization of the DPLL procedure to ZOLCS has already been proposed by Barth [13], its acceleration through lazy clause evaluation for arbitrary pseudo-Boolean constraints is a novel contribution.

To apply lazy clause evaluation to pseudo-Boolean constraints we have to determine a subset of unassigned literals from each constraint such that watching these literals is sufficient to detect that propagations are required to maintain satisfiability of the constraint. Obviously, we are looking for minimal sets with this property in order to avoid unnecessary constraint evaluations. To this end, we arrange the literals in each constraint with respect to their weights, such that the literal with the largest weight is the leftmost one. Then we read the constraint from left to right and select the literals to be watched as follows:

1) The leftmost literal is selected.

2) The following literals are selected until the sum of their weights, not including the weight of the leftmost literal, is greater than or equal to the threshold of the constraint.

The set of literals chosen to be watched by these rules fulfills the above requirements:

a) Assignments which do not affect the watched literals will never make the constraint unsatisfiable as the literals selected by rule 2) are by themselves sufficient to ensure satisfiability.

   Furthermore, no such assignment will turn the constraint into a propagating one. To see this, assume that an assignment to an unwatched literal *does* cause the propagation of some literals. Then, as shown before, the watched literal with the largest weight, selected by rule 1), is among those implications. However, according to the propagation rule from section 3.3.1, this literal is *not* propagated because the remaining watched literals, selected by rule 2), ensure satisfiability of the constraint.

Consequently, a visit of the constraint is unnecessary upon assignments affecting unwatched literals only.

b) If, on the other hand, an assignment sets a watched literal to false, the corresponding constraint may become propagating. Hence, such assignments trigger an evaluation of the constraint. However, the set of watched literals guarantees that the constraint is satisfiable when being visited as any single literal can be assigned while preserving satisfiability of the constraint: if the literal with the largest weight is set to false then the watched literals selected by rule 2) still guarantee satisfiability. If a literal selected according to rule 2) is set to false, the literal selected by rule 1) can compensate for it as its weight is greater than or equal to the one of the literal which has been assigned.

c) The chosen literals form a minimal subset with these properties as we start selecting from the largest weight towards the smallest.

**Example 3.5.** Consider the pseudo-Boolean constraint

$$4a + 3b + 2c + 1d + 1e + 1f + 1g \geq 5$$

whose literals have been ordered with respect to their weights from left to right in descending order. According to rules 1) literal $4a$ is watched, because it is the leftmost one. Furthermore, literals $3b$ and $2c$ are watched according to rule 2), because $3+2 \geq 5$, i.e. the constraint can be satisfied solely by assigning appropriate values to $b$ and $c$.

If a watched literal of a constraint is assigned false, our algorithm tries to reestablish a set of literals which is in accordance with rule 1) and 2). This requires the search for a minimal set of literals which are either unassigned or true and whose weights sum up to a value that at least equals that of the watched literal which has been assigned false. If such a set exists, then it is added to the set of watched literals to replace the one which has dropped out. If no such set exists then this indicates that the constraint has become propagating. The required propagations are determined by application of the propagation rule from section 3.3.1.

In our implementation, we decided to continue to watch those literals which have been assigned true as well as those which have been set to false but cannot be substituted, instead of really removing them from the watch-set. Clearly, these watched literals will not trigger any constraint evaluation as long as the corresponding variables keep their values, s.t. there is no need to really remove them. The desired side-effect

is, however, that upon *un*assignment of the corresponding variables, these watched literals are implicitly reactivated. This form of *lazy reactivation* ensures that after backtracking an appropriate set of literals is watched in each clause. As this is achieved without any computational burden, the time spent on backtracking is linear in the number of unassignments that have to be performed, just as in CNF-SAT. The price to be paid is, however, that after backtracking the watch-set of a clause may no longer be minimal, because a single watched literal, which after backtracking is unassigned again, might have previously been replaced by several literals with smaller weights. In principle we could re-establish a minimal set of watched literals by recording the changes in the watch-sets triggered by each decision and undoing these changes when backtracking. Yet, this would make backtracking much more expensive and, furthermore, we would abandon the advantage that reassigning a variable shortly after unassigning it – due to the nature of tree-search algorithms a quite frequent case – is usually faster than its previous assignment, as thereafter it is only watched in a small subset of constraints.

**Example 3.6.** Consider again the pseudo-Boolean constraint from the previous example. Figure 3.3 on the next page illustrates the changes of the set of watched literals when successively setting $g$, $c$ and $b$ to false, then unassigning all variables.

### 3.3.3 Learning in Presence of Pseudo-Boolean Constraints

Just as in CNF-SAT, conflict-driven learning in our solver for ZOLCS is based on an analysis of the implication graph as described in section 3.2.2.

For sake of efficiency, the implication graph is, exactly as e.g. in the ZCHAFF solver [125], only maintained implicitly during the search by annotating each variable assigned by propagation with a pointer to its propagating constraint, but it is not constructed explicitly until a conflict is really encountered.

To construct the implication graph, the conflict diagnosis procedure has to determine, for each propagation involved in the conflict, a subset of literals of the corresponding propagating constraint whose assignment triggered the propagation. In CNF-SAT, this subset is easy to identify as it consists of *all* literals of the unit clause, except the one which has been propagated. In ZOLCS-SAT, however, we have to take into account that a pseudo-Boolean constraint is not necessarily satisfied after propagation and may even become propagating more than once. Hence, the state of its literals at the time of conflict analysis may not coincide with the state at the time of

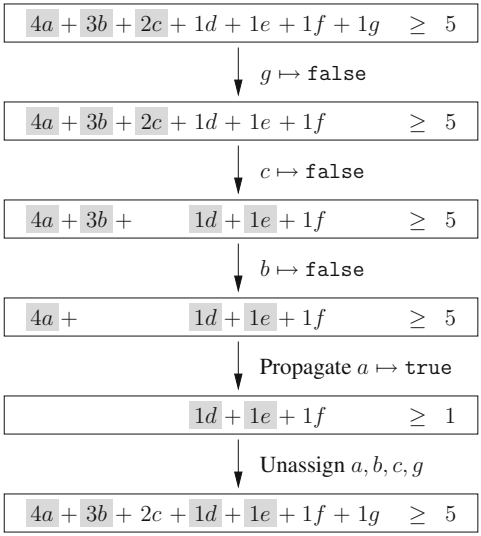


Figure 3.3: Assignments to variables trigger modifications of the watch-set.

propagation. Selecting an arbitrary set of assigned literals from a constraint to reconstruct a cause for a specific propagation may thus introduce acausal relationships or even cycles in the implication graph.

To avoid this, our implementation associates a timestamp with each variable assignment. The timestamp records the point in time when the variable was assigned; it is reset to zero upon unassignment of the variable. Let $\sum_{i \in M} a_i x_i \geq k$, $M \subseteq \mathbb{N}$ be a pseudo-Boolean constraint and $x_p$ a literal propagated by this constraint. For each $i \in M$ let $t_i$ denote the timestamp associated with literal $x_i$. To determine a reason for the propagation of $x_p$, our implementation chooses a minimal subset of literals with indices $R \subseteq M$ such that $(\sum_{i \in M \setminus R} a_i) - a_p < k$ and $(\sum_{i \in M \setminus R} a_i) + a_p \geq k$ hold and, furthermore, for each $r \in R$ the literal $x_r$ has been assigned false with timestamp $t_r \leq t_p$.

The implication graph is built by recursively relating propagated literals to their reasons, where the latter are determined according to the above rules. Once the implication graph has been constructed, the algorithm for conflict analysis and learning is equal to the one used in CNF-SAT. In particular, the clauses learned are CNF clauses.

## 3.4 Benchmark Results

In order to evaluate the proposed methods, we have implemented GOBLIN, a SAT solver for zero-one linear constraint systems, which employs lazy clause evaluation and conflict-driven learning as explained in section 3.3.1. For some experiments we used ZCHAFF, version 2001.2.17, as reference. GOBLIN, which is written in C++, shares several algorithmic features of the ZCHAFF solver, however, as opposed to ZCHAFF, it uses neither random restarts, nor deletion of learned conflict clauses to avoid memory explosion, and it uses a less sophisticated decision strategy. All experiments were performed on a 1 GHz Pentium III machine with 640 MByte physical memory, running Linux.

The first group of experiments dealt with scheduling problems, originally formulated in discrete-time Duration Calculus. The propositional formulae used as input for the SAT engines are bounded model construction problems for Duration Calculus formulae, generated by the method proposed in [59]. Each SAT instance entails the search for a feasible schedule for three periodic tasks, where a feasible schedule corresponds to a satisfying valuation of the respective instance. The individual instances only differ in the runtime $n$ of the tasks involved in the corresponding task system.

The generation of the input formulae for GOBLIN exploits that a duration formula of shape $\int y \geq k$, which holds on an observation interval $O = [m, n]$ iff the accumulated duration of $y$ being true over this interval exceeds $k$, can be directly translated into the threshold clause $\sum_m^n y_i \geq k$, where the $y_i$ represent the values of $y$ in the different time instants.

The results, presented in table 3.2, show that this encoding yields ZOLCS which have size almost constant in $n$, whereas the size of the corresponding CNF-formulae grows rapidly with $n$. The same holds for the solving time. In total, GOBLIN finished all instances in less than three seconds, whereas ZCHAFF required more than 23 minutes to complete them.

The second group of experiments was carried out to assess the effectiveness of the proposed lazy clause evaluation scheme in terms of the average number of clause evaluations that have to be executed after each assignment performed. To this end, we translated integer arithmetic problems into ZOLCS, using a bit-wise encoding that replaces each occurence of an integer variable $a$ with $\sum_{i=0}^{n-1} 2^i a_i$, where the $a_i$ are propositional variables and $n$ is the number of bits needed to represent $a$.

Consider for example the integer constraint $a^2 \geq 4$, where $a$ is a 2-bit integer variable. The encoding sketched above yields $(2a_1 + a_0)(2a_1 + a_0) \geq 4$, i.e. $4a_1^2 + 4a_1a_0 + a_0^2 \geq 4$, which can easily be transformed into a ZOLCS by replacing each

| | CNF | | | ZOLCS | | |
|---|---|---|---|---|---|---|
| $n$ | Clauses | Literals | zChaff $[s]$ | Clauses | Literals | Goblin $[s]$ |
| 1 | 56758 | 140637 | 0.19 | 21638 | 109054 | 0.23 |
| 2 | 107350 | 273069 | 0.40 | 22501 | 108703 | 0.25 |
| 3 | 157942 | 405501 | 1.97 | 21467 | 108193 | 0.25 |
| 4 | 208534 | 537933 | 20.24 | 21386 | 107536 | 0.26 |
| 5 | 259126 | 670365 | 153.56 | 21308 | 106744 | 0.25 |
| 6 | 309718 | 802797 | 163.45 | 21233 | 105829 | 0.25 |
| 7 | 360310 | 935229 | 219.42 | 21161 | 104803 | 0.24 |
| 8 | 410902 | 1067661 | 250.35 | 21092 | 103678 | 0.24 |
| 9 | 461494 | 1200093 | 277.04 | 21026 | 102466 | 0.24 |
| 10 | 512086 | 1332525 | 307.26 | 20963 | 101179 | 0.24 |

Table 3.2: Results of scheduling benchmarks.

| | Formula | Range | Clauses | Literals |
|---|---|---|---|---|
| a) | $a^3 + b^3 + c^3 = d^3$ | $a, b, c, d \in [100; 200]$ | 1103 | 5352 |
| b) | $a^3 + b^3 = c^3$ | $a, b, c \in [100; 200]$ | 827 | 3952 |
| c) | $a^3 + b^3 = 352 * a * b$ | $a, b \in [100; 500]$ | 1018 | 4872 |
| d) | $a^3 + b^3 = 517 * a * b$ | $a, b \in [100; 500]$ | 1018 | 4872 |
| e) | $(416 - a) * b^2 = a^3 \wedge\ a \neq b$ | $a, b \in [100; 500]$ | 855 | 3915 |
| f) | $(224 - a) * b^2 = a^3 \wedge\ a \neq b$ | $a, b \in [100; 500]$ | 855 | 3915 |

Table 3.3: Integer arithmetic problems.

nonlinear term $a_i a_j$ with an auxiliary variable $h$ and imposing the additional constraint $h \rightarrow a_i \wedge a_j$, where the latter is expressed by $2\overline{h} + a_i + a_j \geq 2$.

Table 3.3 shows the integer formulae that were used in the experiments, as well as the size of the corresponding ZOLCS.

As a reference for these benchmarks we used a modified version of GOBLIN that watches *all* literals appearing in a threshold clause, thus mimicking the behaviour of a naive implementation of the DPLL algorithm, that, after each assignment, re-evaluates all constraints containing the literal falsified by that assignment.

The results of the experiments are summarized in table 3.4. 'EpA' denotes the average number of clause evaluations per assignment required in the respective solver run. For all benchmarks performed, lazy clause evaluation is able to significantly reduce this value, as well as the runtime of the solver. The largest gain is obtained

| | Solver Result | Naive Algorithm | | | Lazy Clause Evaluation | | |
|---|---|---|---|---|---|---|---|
| | | Time $[s]$ | Assignments Evaluations | EpA | Time $[s]$ | Assignments Evaluations | EpA |
| a) | $a = 108, b = 114,$ $c = 126, d = 168$ | 94.32 | 2716535 651362897 | 239.78 | 25.88 | 2853381 23950767 | 8.39 |
| b) | UNSAT | 105.25 | 2334859 732186876 | 313.59 | 21.97 | 2412304 24734089 | 10.25 |
| c) | $a = 176, b = 176$ | 38.35 | 1273108 212213069 | 166.69 | 14.12 | 1291765 12169638 | 9.42 |
| d) | UNSAT | 268.96 | 2624583 1268553553 | 483.36 | 37.10 | 2628195 40664770 | 15.47 |
| e) | $a = 288, b = 432$ | 8.27 | 425303 37606297 | 88.42 | 3.68 | 437408 2420138 | 5.53 |
| f) | UNSAT | 19.84 | 622024 98046695 | 157.63 | 6.47 | 647989 5633296 | 8.69 |

Table 3.4: Results of integer problems.

for benchmark d), where EpA is improved by a factor of 31.2, yielding a speed-up factor of 7.2. The reduction of EpA demonstrates that keeping the observation sets on backtracking does not lead to inefficiently large observation sets. It seems that the lower cost of backtracking gained from this strategy in fact outweighs the clause evaluation overhead caused by non-minimal observation sets, as our scheme provides considerable speedups even on general ZOLCS, like the arithmetic benchmarks. In contrast, Chai and Kuehlmann decided, based on benchmarking their implementation, to constrain lazy clause evaluation to CNF and so-called cardinality constraints (i.e. constraints having only weights of 1) only [32].

## 3.5 Discussion

Our experiments indicate that incorporating state-of-the-art SAT solver algorithmics, in particular lazy clause evaluation, into a ZOLCS satisfiability checker is efficient, with the overhead incurred from solving the more expressive source language often being outweighed by the more concise input language. Especially promising are the results obtained on integer arithmetic problems, a domain traditionally considered to be extremely challenging for propositional solvers, be it BDD-based or DPLL-based solvers. It is worth noting that most of the arithmetic operations involved were

nonlinear such that they are out-of-scope of integer linear programming procedures, unless bit-wise encoding is used, which yields poorly performing ILPs in general [98].

The work described in this chapter has been published back in 2003 already. Since then, pseudo-Boolean constraint solving has become an active field of research, as witnessed by the *Pseudo-Boolean Evaluation*, a competition of solvers for pseudo-Boolean satisfiability and optimization problems, first organized in 2005 and held again in 2006, 2007, and 2009. See [86] for a detailed report on the 2005 event.

The solvers, which participated in the competition so far, basically fall into two groups. Adapting techniques from CNF-based SAT checking, solvers in the first group handle pseudo-Boolean constraints *natively*. GALENA [32], BSOLO [87], PBS4 [4], and PUEBLO [110] belong to this group. As our solver does, some of these solvers, e.g. BSOLO and PBS4, learn CNF clauses only. Yet, there are meanwhile also solvers which derive cardinality constraints [32] and general pseudo-Boolean constraints [87, 110] as explanations for conflicts, e.g. by adopting cutting plane techniques from ILP.

Solvers in the second group, among them MINISAT [51], translate pseudo-Boolean constraints (e.g. by making use of BDDs as intermediate format) into CNF clauses and use conventional CNF engines for satisfiability ecking. Pursuing a translation approach, Eén and Sörensson [51] state that their solver *'can perform on par with the best existing native pseudo-Boolean solvers'*, especially if the problem is mostly in CNF and contains few of pseudo-Boolean constraints only. Yet, they believe that domain specific solvers for pseudo-Boolean constraints are *'likely to outperform translation based methods in many cases'*. Indeed, the native solver PUEBLO won the category of instances without optimization function in the 2005 competition. It is worth noting, that PUEBLO was one of only two solvers in the competition which were using lazy data structures, like those proposed in this chapter, not only for CNF clauses, but also for general pseudo-Boolean constraints.

# 4 Integration of DPLL-SAT and Linear Programming

In this chapter, we present a decision procedure which is tailored to fit the needs of BMC of infinite-state systems with piecewise linear variable updates, e.g. of linear hybrid automata, as introduced in chapter 2. Our tool, we name it HYSAT-1, tightly integrates a DPLL style SAT solver with a linear programming routine, combining the virtues of both methods: Linear programming adds the capability of solving large conjunctive systems of linear inequalities over the reals, whereas the SAT solver accounts for fast Boolean search and efficient handling of disjunctions. Building on the work presented in chapter 3, we use our pseudo-Boolean solver GOBLIN as SAT engine in HYSAT-1.

The idea to combine algorithms for SAT with decision procedures for conjunctions of numerical constraints in order to solve arbitrary Boolean combinations thereof has been pursued by several groups. A tight integration of a resolution based SAT checker with linear programming has first been proposed and successfully applied to planning problems by Wolfman and Weld [121]. Tools supporting more general classes of formulae are BARCELOGIC [27], CVC [11], ICS [46], MATHSAT [28], YICES [49] and Z3 [45], all integrating decision procedures for various theories, including Boolean logic, linear real arithmetic, uninterpreted function symbols, functional arrays, and abstract data types, for example.

However, except for HYSAT-1, all tools mentioned above lack some or all of the particular optimizations that arise naturally in the bounded model checking context. As observed by Strichman [113], BMC yields SAT instances that are highly symmetric as they comprise a $k$-fold unrolling of the systems transition relation. This special structure can be exploited to accelerate solving, e.g. by copying the explanation for a conflict which was encountered during the backtrack search performed by the SAT solver, to all isomorphic parts of the formula in order to prune similar conflicts from the search tree. This technique, in the following referred to as *isomorphy inference*, has been shown to yield considerable performance gains when performing BMC with propositional SAT engines. To the best of our knowledge, HYSAT-1 was the first

C. Herde, *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure*, DOI 10.1007/978-3-8348-9949-1_4,

solver that extends isomorphy inference accross transitions, as well as other domain-specific optimizations detailed in [113], to the hybrid domain. Later, these techniques were also employed in the solvers implemented and described by Ábrahám et al. in [1, 2].

We will show that, compared to purely propositional BMC, similar or even higher performance gains can be accomplished within this context. The reason is that an inference step in the hybrid domain is computationally much more expensive than in propositional logic, as now richer logics have to be dealt with. This result is consistent with the findings of Ábrahám et al.

The chapter is organized as follows. In section 4.1 we introduce the logical language solved by our satisfiability checker. Thereafter, we explain in section 4.2 the lazy approach to the satisfiability modulo theory problem, which is the algorithmic basis for our solver. In section 4.3 follows an explanation of our use of linear programming as decision procedure for linear arithmetic. In section 4.4, we discuss the BMC-specific optimizations implemented in our tool. Section 4.5 provides experimental results, and section 4.6 draws conclusions and reviews more recent related work.

## 4.1 The Logics

We address satisfiability problems in a two-sorted logics entailing Boolean-valued and real-valued variables. When encoding properties of hybrid systems, the Boolean variables are used for encoding the discrete state components, while the real variables represent the continuous state components. Aiming at solving formulae which arise as verification conditions in BMC of linear hybrid automata, we have to deal with arbitrary Boolean combinations of propositional variables and linear arithmetic constraints over the reals. The internal format processed by our decision procedure is, however, slightly more restrictive in syntax. The formulae are actually conjunctions of *linear pseudo-Boolean constraints* (as defined on page 39) for the Boolean part and of *guarded linear constraints* for the real-valued part:

$$\begin{aligned} \mathit{formula} &::= \{\mathit{clause} \wedge\}^* \mathit{clause} \\ \mathit{clause} &::= \mathit{linear\_PB\_constraint} \mid \mathit{literal} \rightarrow \mathit{linear\_constraint} \end{aligned}$$

The reason for using pseudo-Boolean constraint clauses instead of ordinary disjunctive clauses (like in CNFs) is that linear pseudo-Boolean constraints are much

more concise than disjunctive clauses and that we have a very efficient SAT solver for pseudo-Boolean constraint systems, yielding the base engine for HYSAT-1. The use of pseudo-Boolean constraints is, however, not essential for the workings of our algorithms; we could use an ordinary CNF solver as well, provided that it is DPLL-based.

To express constraints between real-valued variables, as well as their dependencies with the Boolean valuation, we use guarded linear constraints. A guarded linear constraint is an implication between a Boolean literal and a linear equation or inequation over real-valued variables, the latter denoted by the non-terminal *linear_constraint* in above syntax-definition. The intended semantics is that satisfaction of the guard enforces that the linear constraint must be satisfied, too. We say that satisfaction of the guard *activates* the linear constraint.

A *formula* $\phi$, i.e. a conjunction of pseudo-Boolean constraints and of guarded linear constraints, is interpreted over valuations

$$\sigma = (\sigma_{\mathbb{B}}, \sigma_{\mathbb{R}}) \in (BV \xrightarrow{\text{total}} \mathbb{B}) \times (RV \xrightarrow{\text{total}} \mathbb{R}).$$

Obviously, $\phi$ is satisfied by $\sigma = (\sigma_{\mathbb{B}}, \sigma_{\mathbb{R}})$, denoted $\sigma \models \phi$, iff all linear zero-one constraints in $\phi$ are satisfied by $\sigma_{\mathbb{B}}$ and all guarded linear constraints in $\phi$ are satisfied by $(\sigma_{\mathbb{B}}, \sigma_{\mathbb{R}})$. A guarded linear constraint $v \rightarrow c$ is satisfied by $\sigma = (\sigma_{\mathbb{B}}, \sigma_{\mathbb{R}})$ iff $\sigma_{\mathbb{R}}$ satisfies the linear constraint $c$ or if $\sigma_{\mathbb{B}}(v) = \texttt{false}$.

Like in the chapter 3, we will build valuations incrementally using Davis-Putnam-like procedures and have thus to reason about *partial valuations* $\rho \in (BV \xrightarrow{\text{part.}} \mathbb{B}) \times (RV \xrightarrow{\text{part.}} \mathbb{R})$ of variables. We define the notion of an unassigned variable, as well as the notions consistency, inconsistency and satisfaction for a partial valuation analogously to the definitions given on page 40. Furthermore, we will use the same notation $\rho \models \phi$ for satisfaction by partial and by total valuations.

An *arbitrary* Boolean combination of linear constraints can be easily converted into our internal syntax using a definitional translation as detailed in chapter 3, where linear arithmetic constraints are treated like ordinary subformulae, i.e. definitions are introduced for them. Definitions involving Boolean variables only are converted, in a second translation step, into pseudo-Boolean constraints. We refer to the conjunction of the latter as *Boolean abstraction* of the input formula in the following. All other definition, i.e. those involving arithmetic predicates, are kept (either unmodified or in contraposition) as guarded linear constraints, where the Tseitin labels act as guards.

**Example 4.1.** Figure 4.1 illustrates the definitional, satisfiability-preserving translation of $\phi = (3x + y \geq 4) \vee \neg(b \vee (x < 5))$ into our internal syntax. Like in the

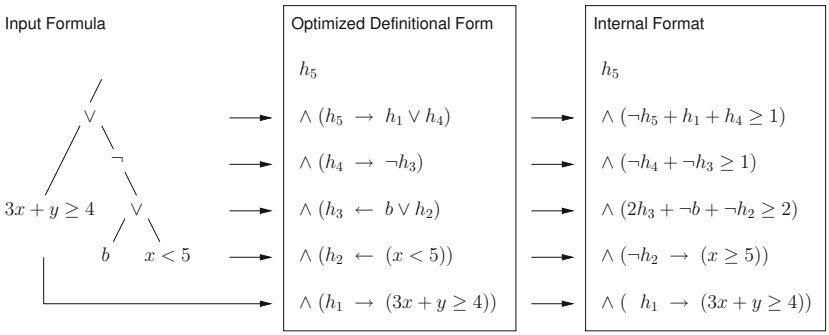


Figure 4.1: Translation into the internal format.

pure propositional case, we use the definitional translation with polarity optimization. Note that the linear constraint $x < 5$ appears in negated form, i.e. as $x \geq 5$, in the guarded constraint, because it occurs with negative polarity in $\phi$.

Furthermore, we apply all optimizations listed on page 44. If there were, for example, a positive occurence of $3x + y < 4$ in $\phi$, then we would use $\neg h_1$ as Tseitin label (and guard) for it.

## 4.2 Lazy Approach to SMT

Our integration of the DPLL procedure and linear programming is an instance of the lazy approach to the Satisfiability Modulo Theories (SMT) problem, which is currently considered to be the most effective approach to SMT [108].

The basic idea of lazy SMT is to combine a DPLL-based SAT solver with a theory solver that can decide the satisfiability of *conjunctions* of atoms over a given theory $T$ in order to obtain a DPLL($T$) solver which can deal with complex Boolean combinations of propositional atoms as well as atoms over $T$.

Figure 4.2 on the facing page depicts the interaction of the two solvers in a DPLL($T$) set-up. The DPLL engine solves a Boolean abstraction of the input formula $\phi$ which is obtained, as demonstrated in section 4.1, by replacing each $T$-atom $c_i$ occuring in $\phi$ with a fresh Boolean variable or its negation, i.e. with a literal $b_i$, the latter serving as guard literal for $c_i$ in the sense that $c_i$ is activated when $b_i$ becomes true.

In lock-step with the incremental assignment of truth values to Boolean variables, the DPLL procedure builds a conjunctive constraint system over $T$ which is given by

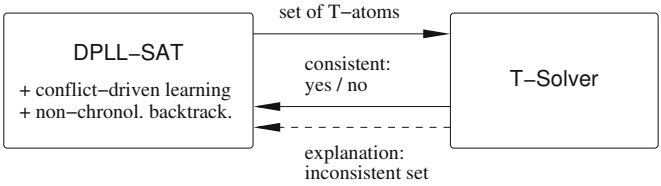


Figure 4.2: Solver interaction in the lazy SMT framework.

the set $\mathcal{C} := \{c_i \mid \rho(b_i) = \texttt{true}\}$ of $T$-atoms being activated by the current partial Boolean valuation $\rho$. This constraint system is passed to a decision procedure for $T$ to decide its consistency, i.e. whether $\mathcal{C}$ is satisfiable or not. If $\mathcal{C}$ is consistent, the DPLL procedure may continue to extend its truth-assignment. In case of inconsistency, a conflict clause is constructed by

- extracting a small (ideally minimal) subset $\mathcal{I}$ from $\mathcal{C}$ which is inconsistent itself and can therefore serve as an explanation for $\mathcal{C}$'s inconsistency,
- building the disjunction $\bigvee_{c_i \in \mathcal{I}} \neg b_i$ which, when added to the clause database, will prevent DPLL from simultaneously activating the conflicting atoms from $\mathcal{I}$ again.

The DPLL solver then performs backjumping such that at least one literal of the conflict clause is freed (i.e. unassigned). By unassigning guard literals due to backjumping, it deactivates $T$-atoms and resolves the conflict. Note that backjumping happens in exactly the same way, irrespective of whether the conflict clause, which controls backjumping, was derived from a propositional conflict or from a $T$-conflict.

## 4.3 SAT Modulo the Theory of Linear Arithmetic

Being interested in solving Boolean combinations of linear arithmetic equalities and inequalities over the reals, we use a DPLL($T$) framework where the $T$-solver is instantiated with a linear programming routine. Linear programming (LP) is used to decide the satisfiability (in this context usually called *feasibility*) of sets of linear constraints, i.e. to check whether for a given system of inequations $\mathbf{A}x \sim b$, where $\sim \in \{\leq, <\}$, the set of solutions $\{x \in \mathbb{R}^n \mid \mathbf{A}x \sim b\}$ is non-empty, as well as a means for efficiently deriving explanations for infeasibility.

Linear programming deals with finding extremal values of a linear objective function when the variables are constrained by linear (in)equations, i.e. with problems that can be put in the general form

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & \mathbf{A}x \leq b \\ & x \in \mathbb{R}^n \end{array} \tag{4.1}$$

where $x$ is the vector of variables to be solved for, $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a maxtrix, and $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$ are column vectors. The linear expression $c^T x$ is called the *objective function*, and (4.1) is referred to as a *linear program*.

Geometrically, the solution set $P = \{x \in \mathbb{R}^n \mid \mathbf{A}\, x \leq b\}$ of the constraint part of a linear program is an intersection of finitely many halfspaces, called *polyhedron*. If a polyhedron is bounded, it is called *polytope*. It can be shown that the maximum of the objective function $c^T x$ is attained at a *vertex* of $P$, provided that $\max\{c^T x \mid x \in P\}$ is not unbounded.

The main reason for preferring LP over other methods of detecting feasibility of linear constraint systems (e.g., Fourier-Motzkin Elimination [58, 92, 23]) is that linear programming is known to be polynomial and scales extremely well in practice, even though the most frequently used codes are actually based on the non-polynomial *Simplex algorithm* [40, 107]. Commercial codes like CPLEX tackle instances with more than $10^6$ variables. In our solver, however, we use the free LP library GLPK[1] by Andrew Makhorin.

To find an optimal vertex of (4.1), the simplex algorithm proceeds as follows. Starting at an initial vertex of $P$, the algorithm checks, whether the vertex has outgoing edges along which the objective function increases. If no such edge exists, then the current node is optimal. Otherwise, the algorithm follows one of these edges. Doing so, it either reaches a neighbouring vertex, where it repeats the same procedure, or the edge is infinite, indicating that the linear program is unbounded. The initial vertex required by the simplex algorithm as starting point is computed by applying the simplex method to the auxiliary linear program

$$\begin{array}{ll} \text{maximize} & \mathbf{1}^T y \\ \text{subject to} & \mathbf{A}x - by \leq \mathbf{0} \\ & \mathbf{0} \leq y \leq \mathbf{1} \\ & x \in \mathbb{R}^n \\ & y \in \mathbb{R}^m \end{array} \tag{4.2}$$

[1] http://www.gnu.org/software/glpk/glpk.html

using $(0, \ldots, 0) \in \mathbb{R}^{m+n}$ as initial vertex. If the optimum value of (4.2) is 0, then (4.1) is infeasible. Otherwise the optimum value is 1 and the solution values for $x$ can be used as initial vertex to solve (4.1). See [107] for a precise algebraic description of the two-phase simplex method sketched above.

### 4.3.1 Feasibility Check Using LP

Checking the feasibility of a system of weak (i.e. non-strict) linear inequations by linear programming is straightforward and requires only a hand-over of the unmodified linear constraint system to the LP solver, plus generation of a trivial objective function.

To cope with systems containing strict inequations, which cannot be handled by LP directly, we use the standard trick of introducing a fresh real-valued variable $\varepsilon$ and of replacing each strict inequation $\sum_{j=1}^{n} \mathbf{A}_{i,j}\, x_j < b_i$ by $\sum_{j=1}^{n} \mathbf{A}_{i,j}\, x_j + \varepsilon \leq b_i$. Instrumenting the resultant linear constraint system with the objective function $\varepsilon$ to be maximized yields an LP which is feasible with strictly positive optimum iff the original constraint system is feasible.

During the search performed by the DPLL solver, the linear constraint system to be checked for feasibility is growing (in phases when DPLL extends a partial solution) and shrinking (upon backjumps) again and again. To avoid the overhead entailed by actually adding (and retracting) linear constraints to (from) the LP, our implementation uses the standard LP mechanism of just activating them on demand by enabling the respective row bounds in the LP, thus allowing the simplex algorithm to restart solving from the dual solution whose feasibility is not affected by changing bounds in the primal LP.

### 4.3.2 Extractions of Explanations

In case of an arithmetic conflict, i.e. if a set $\mathcal{C} = \{c_1, \ldots, c_m\}$ of (strict and non-strict) linear constraints is infeasible, we want to obtain a a subset $\mathcal{I} \subseteq \mathcal{C}$ that is infeasible itself and which is *irreducible* in the sense that any proper subset is feasible. Such an *irreducible infeasible subsystem* (IIS) is a prime implicant of all possible reasons for the unsatisfiability of the constraint system $C$, and is thus a natural counterpart to the conflict clauses in the propositional setting as it prevents the proof search from visiting the same or related inconsistent constraint sets again.

Actually, irreducibility of the subsystem is not a necessity; any infeasible subsystem can serve as explanation. The reason for preferring small subsystems is that these

```
1   DELETION_FILTER(C)
2     for i := 1, ..., m do
3       C := C − {c_i}
4       if is_consistent(C) then
5         C := C ∪ {c_i}
6     return C
```

Listing 4.1: Deletion filter algorithm.

provide more general reasons for conflicts, i.e. reasons which potentially eliminate a bigger number of LP calls.

*Deletion Filtering*

A simple way to isolate an IIS is *deletion filtering* as proposed by Chinnek [34, 33]. Listing 4.1 displays the algorithm in pseudo-code. Given an infeasible system $\mathcal{C} = \{c_1, \ldots, c_m\}$, deletion filtering temporarily removes, one by one, each individual constraint from $\mathcal{C}$ and checks if the reduced system is feasible. If so, the respective constraint is returned to the set; otherwise it is removed permanently. The constraints finally remaining in $\mathcal{C}$ form an IIS. If $\mathcal{C}$ has more than one IIS, then the order in which the constraints are tested determines which IIS is found.

The only prerequisite for implementing a deletion filter is the availability of a consistency check for the given background theory, in our case for linear arithmetic over the reals. No further theoretical insights concerning the background theory are required. Using the feasibility check from the previous section, deletion filtering can thus be applied to systems of linear constraints, irrespective of whether the latter are strict or non-strict. On the negative side, filtering a set of $m$ constraints requires $m$ calls to linear programming. Isolation of an IIS using a deletion filter is thus computational quite expensive.

Some modern SAT engines not only learn reasons for conflicts, they occasionally also *forget* learned clauses, e.g. in order not to clutter memory. In case of pure propositional SAT solving this is acceptable, because a rediscovery of reasons for propositional conflicts is relatively cheap. In view of the high computational cost at which explanations for *arithmetic* conflicts are obtained, it seems however highly recommendable to learn the latter *persistently* and only apply forgetting to conflict clauses derived from propositional conflicts.

*Exploiting Duality*

In case that the constraint system $\mathcal{C}$ contains only non-strict inequations, it is a well-known fact that extraction of irreducible infeasible subsystems can be reduced to finding extremal solutions of a dual system of linear inequations [66, 100]. Given an infeasible system of linear inequalities

$$\begin{cases} \mathbf{A}x & \leq & b \\ x & \in & \mathbb{R}^n \end{cases} \tag{4.3}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$, Gleeson and Ryan [66] use a variant of Farkas' Lemma [56] to obtain a polytope in which each vertex corresponds to an IIS of (4.3). This polytope is defined by the following system of linear constraints:

$$\begin{cases} y^T\mathbf{A} & = & \mathbf{0} \\ y^T b & \leq & -1 \\ y & \geq & \mathbf{0} \\ y & \in & \mathbb{R}^m \end{cases} \tag{4.4}$$

More precisely, the theorem of Gleeson and Ryan states the following: If (4.4) is infeasible, then (4.3) is feasible. If, on the contrary, (4.3) is infeasible, then (4.4) is feasible, and for each vertex $y \in \mathbb{R}^m$ of the polytope (4.4) the set

$$\mathcal{I} = \{\sum_{j=1}^{n} A_{i,j}\, x_j \leq b_i \mid y_i \neq 0\}$$

of linear inequalities identified by the non-zero components of $y$ is an IIS of (4.3). Therefore, checking the feasibility of (4.3) and, in case of infeasibility, locating one of its IIS, both amounts to finding a vertex of a polyhedron, and can thus be solved by linear programming.

In case that the infeasible system $\mathcal{C}$ contains *strict* inequations, we first relax these to non-strict ones. If the relaxed systems $\tilde{\mathcal{C}}$ turns out to be satisfiable, we resort to computing an IIS of the original system using the deletion filter approach. Otherwise, we apply the method of Gleeson and Ryan to $\tilde{\mathcal{C}}$ in order to obtain an IIS $\tilde{\mathcal{I}}$, corresponding to an infeasible, yet not necessarily irreducible subsystem $\mathcal{I}$ of $\mathcal{C}$. To possibly further reduce $\mathcal{I}$, we finally apply a deletion filter. By filtering only the (in general, substantially) reduced subsystem $\mathcal{I}$, we gain considerable performance compared to applying a deletion filter to $\mathcal{C}$ directly.[2]

[2] Actually, in our implementation the final application of the deletion filter is optional, as the subsystems obtained from the relaxed system are often tight enough (i.e., only marginally larger than

### 4.3.3 Learning from Feasible LPs

Besides learning from arithmetic conflicts, HYSAT-1 is also able to perform forward arithmetic inference, thereby deriving new arithmetic facts from feasible sets of linear constraints. Given a feasible set $\mathcal{C} = \{c_1, \ldots, c_m\}$ of linear constraints, HYSAT-1 employs linear programming to determine for each continuous variable $x$ occurring in $\mathcal{C}$ the minimum value $x_{min}$ and the maximum value $x_{max}$ consistent with $\bigwedge_{i=1}^{n} c_i$. If either of these values exists, HYSAT-1 adds the respective bound constraint, i.e. $x \geq x_{min}$ or $x \leq x_{max}$, guarded by a fresh Boolean variable $p$, to its database, together with a propositional clause which is responsible for triggering the activation of the new constraint. To this end, the propositional clause is of form $p_{c_{i_1}} \wedge \ldots \wedge p_{c_{i_k}} \rightarrow p$, where the variables $p_{c_{i_j}}$ are the guard variables of a minimal set of constraints $c_{i_j} \in \mathcal{C}$ whose conjunction implies the new bound constraint. It is noteworthy that the extraction of the minimal set $\{c_{i_j} \mid 1 \leq j \leq k\} \subseteq \mathcal{C}$ does not entail any computational overhead, but is delivered by the LP solver as a byproduct of determining the bound on the respective continuous variable.

When learning a new bound constraint, HYSAT-1 also adds Boolean clauses capturing all propositional dependencies between bound constraints concerning the same continuous variable, i.e. implicative dependencies between bounds as induced by the linear order on the reals. If the solver e.g. learns that in a certain branch of the search tree $x \geq 5$ holds, it will therefore immediately exclude all combination of assignments to guard variables that would cause the activation of bound constraints $x \leq c$ with $c < 5$, thereby considerably pruning the search space.

### 4.3.4 Putting It All Together: a Sample Run

To finally demonstrate the resulting interaction between pseudo-Boolean DPLL-SAT and linear programming within the HYSAT-1 solver, we consider solving of the formula

$$(2e + C + D \geq 2) \tag{4.5}$$
$$\wedge\ (2f + A + B \geq 2) \tag{4.6}$$
$$\wedge\ \neg f + g + e \geq 1 \tag{4.7}$$
$$\wedge\ \neg g + \neg f \geq 1 \tag{4.8}$$
$$\wedge\ 3\neg e + 2g + C + D \geq 3 \tag{4.9}$$

the prime implicants) such that the overhead incurred from deletion filtering is not amortized by the reduction in search space of the SAT procedure.

$$\wedge\, A \rightarrow (4x - 2y \geq 9) \tag{4.10}$$
$$\wedge\, B \rightarrow (2x - 4y \leq -7) \tag{4.11}$$
$$\wedge\, C \rightarrow (x + y \leq 5) \tag{4.12}$$
$$\wedge\, D \rightarrow (x \leq 7) \tag{4.13}$$

consisting of pseudo-Boolean constraints (4.5) to (4.9) and guarded linear constraints (4.10) to (4.13). Variables $x$ and $y$ are real-valued, variables $A, B, C, D, e, f, g$ are of type Boolean. $A, B, C, D$ are furthermore guards for arithmetic constraints.

The backtrack search tree in figure 4.3 on page 71 illustrates the actions performed by the solver. We assume in the following that the tree is traversed in preorder. For convenience, we denote the arithmetic constraints with their guards, i.e. we refer to $(x \leq 7)$ as constraint $D$, for example. Moreover, we write Boolean assignments in form of literals, e.g. $\neg e$ instead of $e \mapsto \texttt{false}$ and $f$ instead of $f \mapsto \texttt{true}$.

Solving starts with assigning $\neg e$ by decision, thereby satisfying clause (4.9) and triggering propagations $C$ and $D$ by clause (4.5) in the subsequent deduction phase. Being guards, $C$ and $D$ activate their associated linear constraints, entailing a feasibility check of the resulting linear constraint system. Because the latter is consistent, the DPLL part continues with the next decision $f$, which satisfies (4.6) and causes propagation of $g$ and $\neg g$ by clauses (4.7) and (4.8), respectively. The Boolean conflict arising from the conflicting propagations is resolved by flipping the last decision, i.e. by assigning $\neg f$, thereby satisfying clauses (4.7) and (4.8) and causing the activation of the linear constraints $A$ and $B$ through propagations by clause (4.6). The subsequent consistency check by linear programming fails, i.e. we have an arithmetic conflict this time. Analysis (using deletion filtering, e.g.) identifies the irreducible infeasible subsystem $\{A, B, C\}$ as explanation, revealing that $D$ is irrelevant for the conflict. In order to avoid checking the conflicting subsystem again, DPLL learns the conflict clause $\neg A + \neg B + \neg C \geq 1$ and proceeds by backtracking, thereby unassigning guards and deactivating linear constraints. By flipping the initial decision to $e$, DPLL satisfies clauses (4.5) and (4.7) and triggers propagation of $g$ by clause (4.9), $\neg f$ by (4.8), then $A$ and $B$ by (4.6). Because constraints $A$ and $B$ are active now, the conflict clause becomes unit and cuts off a conflicting branch in the search tree by propagation of $\neg C$, the latter entailing activation of $D$ through clause (4.9). Thereafter, all Boolean clauses are satisfied and the system of activated linear constraints is found to be consistent by a final feasibility check, i.e. the solver proved satisfiability of the input formula.

Note that linear programming is called only after Boolean deduction is complete. This is because a single call to LP typically requires as much time as multiple thousands of Boolean propagations. It is therefore reasonable to delay LP calls, e.g. until the end of the Boolean deduction phase or until the linear constraint system has substantially changed, and to look for Boolean conflicts first.

## 4.4 Optimizations for BMC

Compared to related tools like ICS which aim at being general-purpose decision procedures suitable for arbitrary formulae, HySAT-1's decision procedure has been tuned to exploit the unique characteristics of BMC formulae.

As observed by Strichman [113], the highly symmetric structure of the $k$-fold unrolling (cf. chapter 2, section 2.2.1) as well as the incremental nature of BMC can both be exploited for various optimizations in the underlying decision procedure. HySAT-1 implements three optimizations which are described below.

*Isomorphy Inference*

The learning scheme employed in propositional SAT solvers accounts for a substantial fraction of the solver's running time as it entails a non-trivial analysis of the implications that led to an inconsistent valuation. The creation of a conflict clause is in general even considerably more expensive in a combined solver like HySAT-1, as the analysis of a conflict involving non-propositional constraints requires the computationally expensive extraction of an IIS.

Isomorphy inference exploits the (almost) symmetric structure of a BMC formula in order to add isomorphic copies of a conflict clause to the problem, thus multiplying the benefit taken from the costly reasoning process which was required to derive the original conflict clause.

The concept is best illustrated using an example. Suppose that the solver has encountered a conflict which yields the conflict clause $\mathcal{C}^0 = (\overline{x}_3^{j_1} \vee x_4^{j_2} \vee x_9^{j_3})$, relating three variables from cycles $j_1$, $j_2$ and $j_3$. The solver then not only adds $\mathcal{C}^0$ to $\phi^k$, but also all possible clauses $\mathcal{C}^i = (\overline{x}_3^{j_1 \pm i} \vee x_4^{j_2 \pm i} \vee x_9^{j_3 \pm i})$, $i = 1, 2, \ldots$, obtained from $\mathcal{C}^0$ simply by index shifting.

Note, however, that BMC is not fully symmetric because of the initialization properties of runs and perhaps the verification goal. This implies that only conflict clauses inferred from facts which are independent from such asymmetric formula parts may

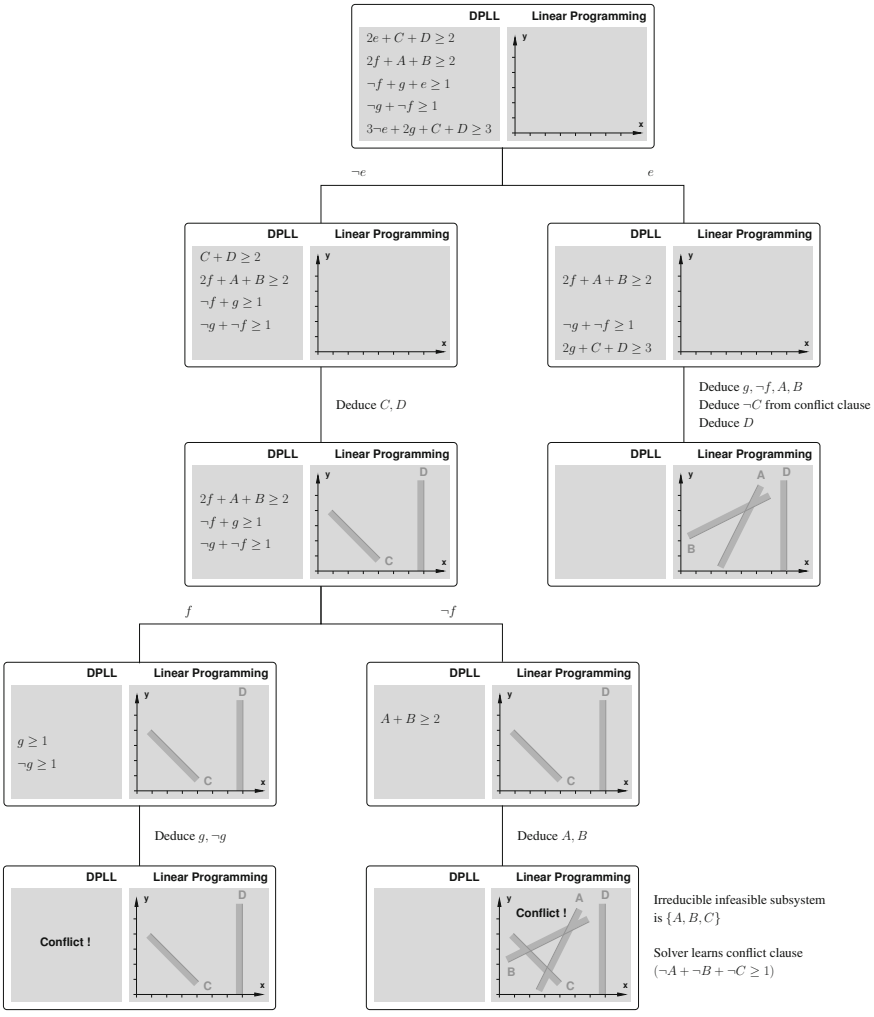


Figure 4.3: Backtrack search tree arising in a tight integration of DPLL proof search with linear programming.

be soundly replicated. Such dependency can be traced cheaply by marking initialization and goal predicates and dominantly inheriting such marks upon all inferences, inhibiting isomorphy inference whenever a mark is encountered.

*Sharing of Conflict Clauses*

When performing BMC incrementally for longer and longer unrollings, the consecutive formulae passed to the solver share a large number of clauses. Thus, when moving from the $k$-instance to the $(k+1)$-instance, we can simply conjoin the conflict clauses derived when solving $k$-instance to the formula for step $k+1$. However, this is only sound for conflict clauses that were inferred from clauses which are common to both instances. We do currently decide this based on simple syntactic criteria, namely that the conflict clause was inferred purely from clauses stemming from the automaton. I.e. the inference may not involve the verification goal, which tends to become a weaker predicate on longer instances, as it usually entails reachability or recurrence. More elaborate schemes have, however, been investigated for propositional BMC in [78].

*Tailored Decision Strategies*

When applying general-purpose decision strategies to BMC formulae one can observe the phenomenon described in [113] that during the SAT search large sets of constraints belonging to distant cycles of the transition relation are being satisfied independently, until they finally turn out to be incompatible, often entailing the need for backtracking over long distances in the search tree.

In HYSAT-1 we adopt the solution proposed by Strichman [113] to avoid this problem: The heuristics of the SAT solver selects the decision variables in the natural order induced by the variable dependency graph of the BMC formula, i.e. either using a *forward strategy*, starting with variables from $\vec{x}^{\,0}$, then from $\vec{x}^{\,1}$, etc., or vice versa, engaging in a *backward strategy*. This allows conflicts to be detected and resolved more locally, speeding up the search, as witnessed by the results shown in figure 4.8.

## 4.5 Benchmark Results

To evaluate the methods proposed above, HYSAT-1's main components are

- the *solver core*, consisting of a tight integration of a SAT solver with a linear programming routine, described in section 4.3, and enhanced with domain-specific optimizations for BMC, as explained in section 4.4,

- an *API* to the solver core, providing methods for formula generation, simplification, common subexpression eliminiation, and for rewriting the resulting formula into a conjunctive form, namely a conjunction of zero-one linear constraints and guarded linear constraints, which is the input format of the solver core,

- a *frontend*, consisting of HySAT-1's input language and a bounded model checker, which performs the unwinding of the transition relation and controls the solver core via API calls.

To fit the needs of BMC, which involves checking the same system on different unrolling depths, the solver core and the API are designed to work in an incremental fashion in the sense that they allow to add (as well as delete) successively sets of constraints to (from) an existing problem and then redo the satisfiability check without starting SAT search from scratch each time.

We conducted a series of experiments on BMC problems of hybrid automata in which we a) compared HySAT-1 with the ICS solver [46], and b) investigated the impact of the individual optimizations by comparing the computation times of our tool when running with and without the respective optimization beeing enabled. The unwindings fed to ICS were obtained through SRI's infinite-state BMC frontend to ICS as distributed in the SAL tool-set [47]. Our benchmarks are

- The *"leaking gas burner"* and *"water-level monitor"* included in the SAL distribution.

- An elastic approach to distance control of trains running on the same track, similar to the car platooning system used in the PATH project. Here, trains can accelerate or decelerate freely if they do not violate their mutual safety envelopes, yet an automatic speed control takes authority over a train if another train gets close, thereby controlling acceleration proportional (within physical limits) to the front and/or back proximity of the neighboring trains.

- A hybrid model of a car equipped with robotized five-speed transmission and a cruise control system which aims at maintaining a certain preset speed by actuating throttle and brake using two PI controllers. We adopted the model as reported by Torrisi in [117] and modified it by adding a realistic clutch behaviour in the initial acceleration phase.

The results of our experiments are shown in figures 4.4 – 4.8, with each data point representing a single BMC instance solved by two engines. Points lying on the diagonal, which is drawn as a solid line in all figures, indicate equal running times of both tools; points lying above (below) the diagonal represent instances that were solved faster by the engine whose running times can be read off from the x-axis (y-axis). Note the logarithmic scaling of the axes in figures 4.4 and 4.5.

It can be seen that the individual optimizations yield consistent performance benefits, with the merits becoming more evident with increasing unrolling depth, corresponding to computationally more costly SAT instances.

With respect to the decision strategy it turns out that there is no single optimal strategy. Depending on the specific shape of the initial state set and the target region, forward or backward strategies, though in general both better than the standard strategy, may be more beneficial. We are experimenting with randomized approaches to on-the-fly strategy switch to overcome the problem of selecting an appropriate strategy a priori.

## 4.6 Discussion

The benchmarks performed indicate a very competitive performance of HySAT-1 when used for bounded model checking of linear hybrid systems. They do thus provide evidence for the effectiveness of HySAT's basic design decisions, which are

1. the exploitation of structural properties of the formulae arising in BMC and
2. the use of a non-clausal and thus more concise base logics.

With the current case studies, which are reachability properties in hybrid automata, measures of the first kind clearly have the predominant effect. Yet our experiments with bounded model construction for the metric-time temporal logic Duration Calculus (see chapter 3, section 3.4) provide evidence that the conciseness gain from using linear zero-one constraint systems instead of CNF formulae will be essential to tractability once observers for metric-time temporal-logic formulae come into play [55].

HySAT-1's techniques for exploiting the particular structure of the verification conditions arising in bounded model checking include inheritance of inference results along the temporal axis within an BMC instance, sharing of inference results across BMC instances, and decision heuristics in the SAT-solver that pay attention

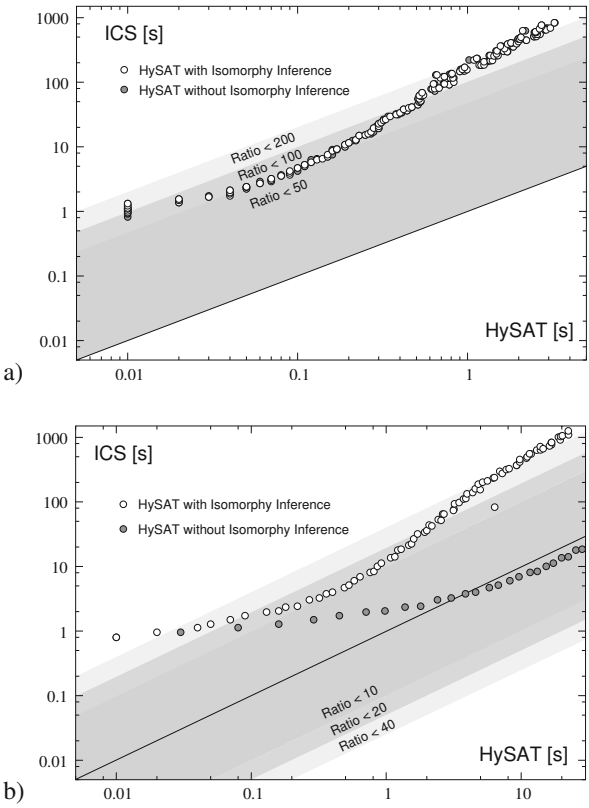


Figure 4.4: Performance of HYSAT-1 relative to ICS: BMC times for a) gasburner model, b) water-level monitor.

to the causal relationship between problem variables by doing chaining along the transition sequence. These algorithms have been inspired by similar optimizations developed by Strichman for finite-state BMC [113]; however such optimizations exhibit an even better payoff on the two-sorted logics used here, as the price for copying inferences increases only marginally while the computational cost of the individual inference grows dramatically in the hybrid-state case. Consequently, the individual optimization yield speedups of up to, and sometimes even considerably exceeding, an order of magnitude. An interesting aspect of isomorphically copying inference re-

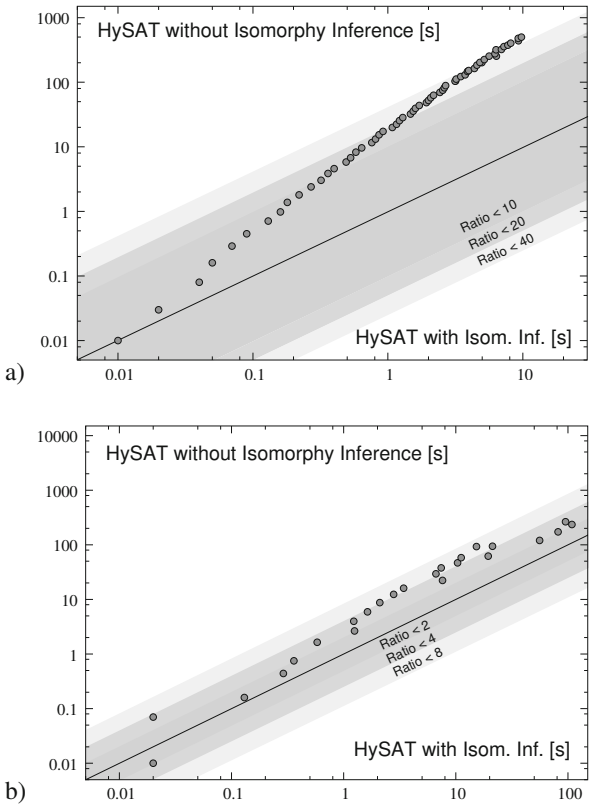


Figure 4.5: Impact of isomorphy inference: BMC times for a) water-level monitor, b) train distance control model, involving 5 trains.

sults, as in inheritance along the temporal axis or in sharing across BMC instances, is that even extremely costly inferences may amortize, provided that their results can be reused sufficiently often. It seems thus worthwile to investigate more general forms of isomorphy inference, e.g. copying inference results across similar components in a multi-component system. While this is in principle similar to exploiting temporal symmetries, the possible forms of symmetry breaks are more diverse and harder to detect, as witnessed by the extensive research on symmetry reduction.

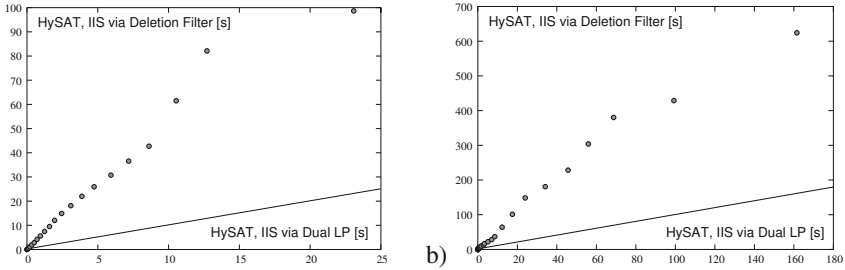


Figure 4.6: Comparison of deletion filter method for extraction of irreducible infeasible subsystems with method using the dual LP. Graphics show results for a) train distance control model, b) car model.

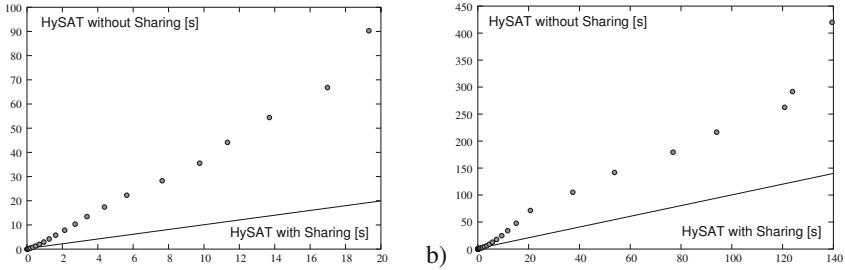


Figure 4.7: Impact of constraint sharing on BMC runtimes for a) train distance control model, b) car model.

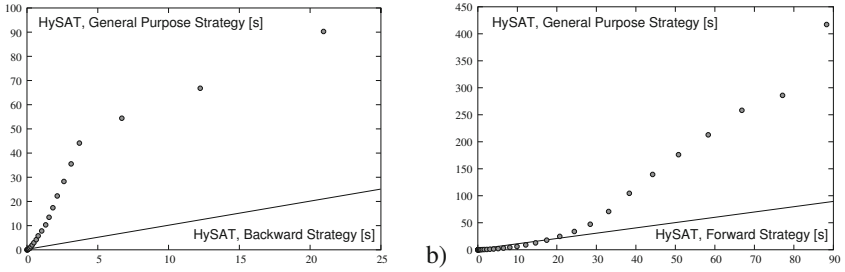


Figure 4.8: Impact of tailored decision strategies: a) For the train model the backward strategy clearly outperforms the general purpose decision strategy, whereas a forward scheme (not shown) slows down the solver. b) Conversely, for the car model the forward strategy is superior.

When we started to work on the integration of linear programming and SAT solving, we deliberately decided to use existing LP solvers instead of implementing an LP solver ourselves. Given the long evolution such tools have undergone and given the huge amount of work which has been done in this field, this seemed to be a reasonable choice. We had to learn the hard way, however, that due to the imprecise floating-point arithmetic used in most LP solvers, the answers given by off-the-shelf LP solvers may simply be wrong. Dhiflaoui et al. state in [48]:

> *'Existing LP-solvers do not claim to solve LPs to optimality. In fact, they come with hardly any guarantee. Feasible problems may be classified as infeasible and vice versa, a solution returned is not guaranteed to be feasible, and the objective value returned comes with no approximation guarantee.'*

Even professionally maintained LP solvers like CPLEX occasionally fail on innocent looking problems, as demonstrated by Neumaier and Shcherbina [95] and by Faure et al. [57]. For applications in formal verification, where correctness of the solver is essential for the validity of the verification results, this is certainly not acceptable. In HYSAT-1 we have therefore implemented various forms of result checking for LP, complemented with strategies for recovery in case of errors, in order to ensure correctness of results. For example, we validate feasibility results, where needed, by checking whether the solution values provided by LP actually satisfy all active linear constraints. Similarly, infeasibility results are checked by validating the solution of the dual (and thus feasible) problem. Yet in spite of such precautions, numerical problems within the LP solver remained a source of instability in HYSAT-1. Another disadvantage of using LP as black-box method is the overhead entailed by solving an optimization problem where only a consistency check is required.

An elegant solution to both problems was presented by Dutertre and de Moura [49, 50] in 2006, two years after we published the work presented in this chapter. For solving systems of linear inequalities within their SMT solver YICES, they developed a variant of the simplex algorithm, called *general simplex* in [83], which performs all computations in exact rational arithmetic and which eliminates the objective function from LP, thereby saving the second (i.e. optimization) phase of the two-phase simplex algorithm. The latter is enabled by handling strictness of inequalities symbolically within the simplex routine, rather than using the objective function to enforce strictness of constraints, as we do. The runtime penalty incurred by using exact arithmetic is apparently more than compensated by saving the optimization phase. The authors

also provide a very efficient solution to the problem of identifying explanations for infeasibility, which are in fact generated as a byproduct of the feasibility check.

Faure et al. [57] present experimental results which indicate, too, that state-of-the-art LP solvers, built for applications in operations research, tend to be less performant than specialized exact solvers when applied in a DPLL($T$) framework. They conjecture that the performance of such specialized solvers can be further improved by implementing them in *inexact* arithmetic and add result checking and error recovery policies, as we did in HYSAT-1, to guarantee correctness of results.

# 5 Integration of DPLL and Interval Constraint Solving

Having dealt with Boolean combinations of *linear* arithmetic constraints in the previous chapter, we now address the problem of solving Boolean combinations of *nonlinear* arithmetic constraints which may contain transcendental functions, like sine, cosine, and the exponential function. This gives rise to a plethora of problems, in particular (a) how to efficiently and sufficiently completely solve conjunctive combinations of constraints in the undecidable domain of nonlinear constraints involving transcendental functions and (b) how to efficiently maneuver the large search spaces arising from the potentially rich Boolean structure of the overall formula.

While promising solutions for these two individual sub-problems exist, it seems that their combination has hardly been attacked. Arithmetic constraint solving based on interval constraint propagation (e.g., [41, 17]), on the one hand, has proven to be an efficient means for solving conjunctions of nonlinear (in particular transcendental) constraints, provided the latter are robust[1] in the sense that their truth value does not change under small perturbations of the occuring constants [104]. Modern SAT solvers, on the other hand, can efficiently find satisfying valuations of very large propositional formulae (cf. chapter 3), and, using the DPLL($T$) framework, of complex propositional combinations of atoms from various decidable theories (cf. chapter 4).

In this chapter, we describe a tight integration of SAT-based proof search with interval-based arithmetic constraint propagation, thus providing an algorithm that reasons over the undecidable formula class of Boolean combinations of nonlinear constraints involving transcendental functions. Undecidability of the arithmetic base theory, however, precludes a DPLL($T$)-style integration. Instead, we exploit the algorithmic similarities between DPLL-based propositional SAT solving and constraint solving based on constraint propagation for a much tighter integration, where the

[1] According to [104], robustness is a property which characterizes *'exactly the problems that model real-life problems in a meaningful way'*.

C. Herde, *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure*, DOI 10.1007/978-3-8348-9949-1_5,

DPLL solver directly manipulates theory atoms instead of a propositional abstraction of the input formula. It has full introspection into and control over constraint propagation within the theory $T$, and it directly integrates any new theory atoms generated by the constraint propagation into the search space of the DPLL solver. This tight integration has a number of advantages. First, by sharing the common core of the search algorithms between the propositional and the theory-related, interval-constraint-propagation-based part of the solver, we are able to transfer algorithmic enhancements from one domain to the other: in particular, we thus equip interval-based constraint solving with all the algorithmic enhancements that were instrumental to the enormous performance gains recently achieved in propositional SAT solving, like watched-literal schemes or conflict-driven learning based on implication-graph analysis. Second, the introspection into the constraint propagation process allows fine-granular control over the necessarily incomplete arithmetic deduction process, thus enabling a stringent extension of SMT to an undecidable theory. Finally, due to the availability of learning, we are able to implement an almost lossless restart mechanism within an interval-based arithmetic constraint propagation framework.

In fact, our integration of DPLL and interval-based constraint solving gives a clean generalization of the DPLL routine to Boolean combinations of arithmetic constraints, which we refer to as iSAT algorithm. iSAT inherits the branch-and-deduce framework with all its recent enhancements from DPLL. For Boolean deduction it uses the unit-propagation rule from DPLL and adds deduction rules for arithmetic operators taken from interval constraint solving. Due to the undecidability of the arithmetic domain, iSAT is necessarily incomplete. This incompleteness manifests itself in the fact that, strictly speaking, only UNSAT results delivered by iSAT are reliable. In practice, however, this turns out to be no serious restriction, because if unsatisfiability cannot be established, then iSAT returns a small box in the search space which it cannot prune any further by deduction and which therefore is likely to either contain genuine solutions or at least *almost-solutions* which violate arithmetic constraints at most by a very small amount.

*Related Work*

In constrast to solvers adopting the DPLL($T$) approach to SMT, our algorithm does not feature the typical two-layered architecture of such tools, which consists of a DPLL-based SAT solver and a subordinated theory solver. Instead, both engines are inseparably interwoven in our algorithm and by this constitute a generalized form of the DPLL routine itself.

From the extensive literature on SMT techniques, the approach coming closest to ours is the *splitting-on-demand technique* in DPLL($T$) of Barett, Nieuwenhuis, Oliveras, and Tinelli [10]. There, the set of theory atoms manipulated by the DPLL solver is made dynamic by a special split rule extending the formula with a tautologous clause introducing new theory atoms. In contrast to this, our tighter integration does not need such helper mechanisms modifying the formula, allows new theory atoms to be generated by both splits and constraint propagations in the theory, and due to its direct manipulation of theory atoms generates atoms on-the-fly and locally to the different branches of the proof-search tree. We conjecture that the more direct integration helps the algorithm to perform stably even under the enormous number of new theory atoms thus being generated. A further crucial distinction to DPLL($T$) is that our algorithm distinguishes between a large (and undecidable) theory that theory propagation acts on (nonlinear arithmetic including transcendental functions) and a small kernel thereof used in consistency checks (real-valued inequation systems). In DPLL($T$) approaches, the roles are generally reversed: consistency check has to cover the full theory $T$, while theory propagation (forward inference) may be more confined, covering a subset of $T$ only, up to being completely missing.

Jussien's and Lhomme's *dynamic domain splitting* technique for numeric constraint satisfaction problems (numeric CSPs) [79] is related to our approach in that both implement conflict-driven learning and non-chronological backtracking within arithmetic constraint solving based on domain splitting and arithmetic constraint propagation. Their approach extends dynamic backtracking (cf. [65]) and provides filtering algorithms for domain reductions (i.e., constraint propagation) enhanced by nogood learning and non-chronological backtracking. Nevertheless, the algorithm described in [79] is not general enough for our problem domain, as it focuses on conjunctive constraint systems. Our algorithm relaxes that limitation and handles nonlinear arithmetic constraint systems with an arbitrary, complex Boolean structure. Another technical difference to [79] is the procedure applied for learning conflicts and the shape of conflict clauses thus obtained. The explanations of conflicts in [79] are confined to be sets of *splitting constraints*, i.e. of choice points decided in branch steps of the branch-and-reduce algorithm. Our algorithm is able to generate more compact and more general conflict clauses entailing both splitting constraints and arbitrary deduced constraints. Like in modern propositional SAT solvers, this is achieved by maintaining and analyzing an implication graph (cf. sect. 5.3.5) storing the immediate reasons for each deduction. We are thus able to generalize all techniques for determining con-

flict clauses which have proven beneficial within propositional SAT, e.g. the 1 UIP learning scheme [125].

Sharing our goal of checking satisfiability of large and complex-structured Boolean combinations of nonlinear arithmetic constraints, Bauer, Pister, and Tautschnig have recently presented the ABSOLVER tool [14]. ABSOLVER is an SMT solver addressing a blend of Boolean and polynomial arithmetic constraint problems. It is an extensible and modular implementation of the SMT scheme which permits integration of various subordinate solvers for the Boolean, linear, and nonlinear parts of the input formula. ABSOLVER itself coordinates the overall solving process and delegates the currently active constraint sets to the corresponding subordinate solvers. The currently reported implementation [14] uses the numerical optimization tool IPOPT (https://projects.coin-or.org/Ipopt) for solving the nonlinear constraints. Consequently, it may produce incorrect results due to the local nature of the solver, and due to rounding errors. Nonetheless, even though in our method we implement strictly correct solving of nonlinear constraints, benchmarks reported in sect. 5.4.2 show that our tighter integration consistently outperforms ABSOLVER, usually by orders of magnitude when formulae with non-trivial Boolean structure are involved. Furthermore, our solver uses interval constraint propagation to address a larger class of formulae than polynomial constraints, admitting arbitrary smooth functions in the constraints, including transcendental ones.

Compared to *interval constraint solving* (ICS, for a survey cf. [17]), our approach is complementary: the interval constraint solving community is primarily concerned with solving — often in the sense of 'paving' the solution set — intricate conjunctive nonlinear constraint systems, and thus concentrates on powerful constraint propagation operators. Our focus is on satisfiability tests for extremely large formulae featuring a complex Boolean structure, which we make feasible by mechanisms for tracking and exploiting the dependencies between subformulae within an SMT framework. Thus, our approach could easily be enhanced by importing more powerful constraint propagation operators, while our mechanisms for maneuvering through large Boolean combinations of nonlinear constraint systems are a contribution to interval constraint solving.

*Structure of the Chapter*

In section 5.1, we expose the syntax and semantics of the arithmetic satisfiability problems our algorithm addresses. Section 5.2 provides a brief introduction to interval constraint solving, which our development builds on. Thereafter, we provide a

detailed explanation of our new algorithm in section 5.3. After reporting on benchmarks conducted to evaluate various aspects of the iSAT algorithm in section 5.4, we demonstrate its application to a case study from the railway domain in section 5.5. In particular, we describe the complete workflow of analyzing the dynamic behaviour of a safety-critical system with our HySAT-2 solver, which implements the iSAT algorithm. Finally, section 5.6 discusses the results achieved.

## 5.1 The Logics

Aiming at automated analysis of hybrid systems, our constraint solver addresses satisfiability of nonlinear arithmetic constraints over real-valued variables plus Boolean variables for encoding the control flow. The user thus may input constraint formulae built from quantifier-free constraints over the reals and from propositional variables using arbitrary Boolean connectives. The atomic real-valued constraints are relations between potentially nonlinear terms involving transcendental functions, like $\sin(x+\omega t)+ye^{-t} \leq z+5$. By the front-end of our constraint solver, these constraint formulae are rewritten to equisatisfiable quantifier-free formulae in conjunctive form, where arithmetic constraints are decomposed into a form resembling three-address code.[2] The *internal* syntax[3] of the constraint formulae processed by our solver is

$$\begin{aligned}
\mathit{formula} &::= \{\mathit{constraint}\ \wedge\}^*\ \mathit{constraint} \\
\mathit{constraint} &::= \mathit{clause} \mid \mathit{definition} \\
\mathit{clause} &::= (\{\mathit{bound}\ \vee\}^*\ \mathit{bound}) \\
\mathit{bound} &::= \mathit{variable}\ \mathit{relation}\ \mathit{rational_const} \\
\mathit{definition} &::= \mathit{variable}\ \mathit{relation}\ \mathit{term} \\
\mathit{term} &::= \mathit{operator}(\{\mathit{variable},\}^*\ \mathit{variable}) \\
\mathit{variable} &::= \mathit{real_variable} \mid \mathit{boolean_variable} \\
\mathit{relation} &::= < \mid \leq \mid = \mid \geq \mid >
\end{aligned}$$

where $\mathit{operator}$ stands for operation symbols, like $+$, $-$, $\times$, $\sin$, and $\mathit{rational\_const}$ ranges over the rational constants. To simplify the exposition, we assume in the remainder of this section that definitions are always equalities and that $\mathit{operator} ::= \mathit{uop} \mid \mathit{bop}$, where $\mathit{uop}, \mathit{bop}$ are unary and binary operation symbols. We refer to

[2] Confer section 5.3.1 for details on the formula conversion.

[3] For examples of the user-level syntax, consult the benchmark files and the manual on the HySAT-2 website http://hysat.informatik.uni-oldenburg.de.

definitions with unary operators as *pairs* and to definitions with binary operators as *triplets*.

While in the syntax given above only Boolean and real-valued variables are allowed, the ISAT algorithm can be easily extended to also handle variables of type integer. In fact, our solver HYSAT-2 implements this extension. For sake of clarity, we decided, however, to restrict our presentation to formulae containing variables of type real and Boolean only. Constraint formulae of above syntax are thus interpreted over valuations $\sigma \in (BV \xrightarrow{\text{total}} \mathbb{B}) \times (RV \xrightarrow{\text{total}} \mathbb{R})$, where $BV$ is the set of Boolean and $RV$ the set of real-valued variables. $\mathbb{B}$ is identified with the subset $\{0,1\}$ of $\mathbb{R}$, so that literals $v$ and $\neg v$ can be encoded by appropriate rational-valued bounds, e.g. $v \geq 1$ or $v \leq 0$. The definition of satisfaction is standard: a constraint formula $\phi$ is satisfied by a valuation iff all its clauses are satisfied, that is, iff at least one atom is satisfied in any clause. Satisfaction of atoms is w.r.t. the standard interpretation of the arithmetic operators and the ordering relations over the reals. In order to make all arithmetic operators total, we extend their codomain (as well as, for compositionality, their domain) with a special value $\mho \notin \mathbb{R}$ such that the operators manipulate values in $\mathbb{R}^{\mho} = \mathbb{R} \cup \{\mho\}$. The comparison operations on $\mathbb{R}$ are extended to $\mathbb{R}^{\mho}$ in such a way that $\mho$ is incomparable to any real number, that is, $c \not\sim \mho$ and $\mho \not\sim c$ for any $c \in \mathbb{R}$ and any relation $\sim \in \{<, \leq, =, \geq, >\}$.

Instead of real-valued valuations of variables, our constraint solving algorithm manipulates interval-valued valuations $\rho \in (BV \xrightarrow{\text{total}} \mathbb{I}_{\mathbb{B}}) \times (RV \xrightarrow{\text{total}} \mathbb{I}_{\mathbb{R}})$, where $\mathbb{I}_{\mathbb{B}} = 2^{\mathbb{B}}$ and $\mathbb{I}_{\mathbb{R}}$ is the set of convex subsets of $\mathbb{R}^{\mho}$.[4] Slightly abusing notation, we write $\rho(l)$ for $\rho_{\mathbb{I}_{\mathbb{B}}}(l)$ when $\rho = (\rho_{\mathbb{I}_{\mathbb{B}}}, \rho_{\mathbb{I}_{\mathbb{R}}})$ and $l \in BV$, and similarly $\rho(x)$ for $\rho_{\mathbb{I}_{\mathbb{R}}}(x)$ when $x \in RV$. In the following, we occasionally use the term *box* synonymously for interval valuation. If both $\sigma$ and $\eta$ are interval valuations then $\sigma$ is called a *refinement* of $\eta$ iff $\sigma(v) \subseteq \eta(v)$ for each $v \in BV \cup RV$.

In order to lift a binary operation $\circ$ and its partial inverses to sets, we define

$$\begin{aligned} m \bullet_1 n &= \{x \mid \exists y \in m, z \in n : x = y \circ z\}, \\ m \bullet_2 n &= \{y \mid \exists x \in m, z \in n : (x = y \circ z \vee \forall y' \in \mathbb{R} : (x \neq y' \circ z \wedge y = \mho))\}, \\ m \bullet_3 n &= \{z \mid \exists x \in m, y \in n : (x = y \circ z \vee \forall z' \in \mathbb{R} : (x \neq y \circ z' \wedge z = \mho))\}, \end{aligned}$$

and similarly for unary $\circ$:

$$\begin{aligned} \bullet_1 m &= \{x \mid \exists y \in m : x = \circ y\}, \\ \bullet_2 m &= \{y \mid \exists x \in m : (x = \circ y \vee \forall y' \in \mathbb{R} : (x \neq \circ y' \wedge y = \mho))\}. \end{aligned}$$

[4] Note that this definition covers the open, half-open, and closed intervals over $\mathbb{R}$, including unbounded intervals, as well as the union of such intervals with $\{\mho\}$.

These are essentially the images of the argument sets under the relation $\{(x, y, z) \mid x = y \circ z\}$ (or $\{(x, y) \mid x = \circ y\}$, resp.) when substituting the respective arguments. We lift these set-valued operators to (computer-representable) intervals by assigning to each set-valued operation $\bullet$ a conservative interval approximation $\hat{\bullet}$ which satisfies $i_1 \hat{\bullet} i_2 \in \mathbb{I}_\mathbb{R}$ and $i_1 \hat{\bullet} i_2 \supseteq i_1 \bullet i_2$ for all intervals $i_1$ and $i_2$ [90]. Note that the definition of an interval extension does not specify how to exactly lift a set operation $\bullet$ to intervals, but leaves some design choice by permitting arbitrary overapproximations. For the sake of reasoning power, $i_1 \hat{\bullet} i_2$ should be chosen such that it provides an as tight as possible overapproximation of $i_1 \bullet i_2$. This means that in practice $i_1 \hat{\bullet} i_2$ is the *interval hull* of $i_1 \bullet i_2$, that is, $\bigcap_{i \in \mathbb{I}_\mathbb{R}, i \supseteq i_1 \bullet i_2} i$ is extended by some outward rounding to compensate for the imprecision of computer arithmetic and the finiteness of the set of floating-point numbers. Now we define a notion that models the fact that an interval valuation fulfills certain subset relations imposed by a formula, namely, those subset relations that the reasoning steps of our algorithm will try to enforce.

For the manipulated interval valuations we adapt the common notion of *hull consistency* (cf. [17]) from interval constraint propagation (cf. sect. 5.2) which our algorithm will try to enforce by reasoning steps. We call an interval valuation $\rho$ *hull consistently satisfying* for a constraint formula $\phi$, denoted $\rho \models_{\text{hc}} \phi$, iff each clause of $\phi$ contains at least one hull consistently satisfied bound and all definitions (triplets and pairs) are hull consistently satisfied. *Hull consistent satisfaction* of atoms is defined as follows:

$$
\begin{array}{lll}
\rho \models_{\text{hc}} x \sim c & \text{iff} & \rho(x) \subseteq \{u \mid u \in \mathbb{R}, u \sim c\} \quad \text{for} \quad x \in RV \cup BV, c \in \mathbb{Q} \\
\rho \models_{\text{hc}} x = y \circ z & \text{iff} & \rho(x) \subseteq \rho(y) \hat{\bullet}_1 \rho(z), \\
 & & \rho(y) \subseteq \rho(x) \hat{\bullet}_2 \rho(z), \\
 & & \rho(z) \subseteq \rho(x) \hat{\bullet}_3 \rho(y) \qquad \text{for} \quad x, y, z \in RV, \circ \in bop \\
\rho \models_{\text{hc}} x = \circ y & \text{iff} & \rho(x) \subseteq \hat{\bullet}_1 \rho(y), \\
 & & \rho(y) \subseteq \hat{\bullet}_2 \rho(x) \qquad \text{for} \quad x, y \in RV, \circ \in uop.
\end{array}
$$

We call a formula $\phi$ *hull consistently satisfiable*, denoted $\text{hcsat}(\phi)$, iff there is an interval valuation $\rho$ with $\rho \models_{\text{hc}} \phi$ and $\rho(v) \neq \emptyset$ for all $v \in BV \cup RV$. Note that hull consistent satisfiability is a necessary, yet not sufficient condition for real-valued satisfiability, as can be seen from the example $(x = x \cdot x) \wedge (x > 0) \wedge (x < 1)$, which is hull consistently satisfied by $\rho(x) = (0, 1)$, yet not satisfiable over the reals.

When solving satisfiability problems of formulae with Davis-Putnam-like procedures, we will build interval valuations incrementally by successively contracting intervals through constraint propagation and branching. This may lead to situations

where an interval valuation does no longer contain any solution, in which case we have to revert some branching decisions previously taken. In order to detect this — in general undecidable — situation, we define a sufficient criterion for non-existence of a solution within the interval valuation: We say that an interval valuation $\rho$ is *inconsistent with an atom* $a$, denoted $\rho \sharp a$, iff the left- and right-hand sides of the atom have disjoint valuations under $\rho$, i.e.

$$\begin{array}{lll} \rho \sharp x \sim c & \text{iff } \rho(x) \cap \{u \mid u \in \mathbb{R}, u \sim c\} = \emptyset & \text{for } x \in RV \cup BV, c \in \mathbb{Q} \\ \rho \sharp x = y \circ z & \text{iff } \rho(x) \cap \rho(y) \hat{\bullet}_1 \rho(z) = \emptyset & \text{for } x, y, z \in RV, \circ \in bop \\ \rho \sharp x = \circ y & \text{iff } \rho(x) \cap \hat{\bullet}_1 \rho(y) = \emptyset & \text{for } x, y \in RV, \circ \in uop \end{array}$$

Note that deciding inconsistency of an interval valuation with an atom (and hence, with a clause or a formula) is straightforward, as is deciding hull consistent satisfaction of an atom (clause, formula) by an interval valuation. If $\rho$ is neither hull consistently satisfying for $\phi$ nor inconsistent with $\phi$ then we call $\phi$ *inconclusive on* $\rho$, which is again decidable.

## 5.2 Algorithmic Basis

Our constraint solving approach builds upon the well-known techniques of interval constraint propagation (ICP) and of propositional SAT solving by the DPLL procedure plus its more recent algorithmic enhancements. An introduction to DPLL-based SAT solving was already given in chapter 3. In the following we give a brief account on ICP, focusing on those aspects which are relevant for understanding the remainder of the chapter.

Interval constraint propagation is one of the sub-topics of the area of constraint programming where constraint propagation techniques are studied in various, and often discrete, domains. For the domain of the real numbers, given a constraint $\phi$ and a floating-point box $B$, so-called *contractors* compute another floating-point box $C(\phi, B)$ such that $C(\phi, B) \subseteq B$ and such that $C(\phi, B)$ contains all solutions of $\phi$ in $B$ (cf. the notion of *narrowing operator* [18, 16]). There are several methods for implementing such contractors. The most basic method [41, 36] decomposes all atomic constraints (i.e., constraints of the form $t \geq 0$ or $t = 0$, where $t$ is a term) into conjunctions of so-called primitive constraints resembling three-address code (i.e., constraints such as $x + y = z$, $xy = z$, $z \in [\underline{a}, \overline{a}]$, or $z \geq 0$) by introducing additional auxiliary variables (e.g., decomposing $x + \sin y \geq 0$ to $\sin y = v_1 \wedge x + v_1 = v_2 \wedge v_2 \geq 0$). Then it applies a contractor for these primitive constraints until a fixpoint is reached.

We illustrate contractors for primitive constraints using the example of a primitive constraint $x + y = z$ with the intervals $[1, 4]$, $[2, 3]$, and $[0, 5]$ for $x$, $y$, and $z$, respectively: We can solve the primitive constraint for each of the free variables, arriving at $x = z - y$, $y = z - x$, and $z = x + y$. Each of these forms allows us to contract the interval associated with the variable on the left-hand side of the equation: Using the first solved form we subtract the interval $[2, 3]$ for $y$ from the interval $[0, 5]$ for $z$, concluding that $x$ can only be in $[-3, 3]$. Intersecting this interval with the original interval $[1, 4]$, we know that $x$ can only be in $[1, 3]$. Proceeding in a similar way for the solved form $y = z - x$ does not change any interval, and finally, using the solved form $z = x + y$, we can conclude that $z$ can only be in $[3, 5]$. Contractors for other primitive constraints can be based on interval arithmetic in a similar way. There is extensive literature [93, 75] providing precise formulae for interval arithmetic for addition, subtraction, multiplication, division, and the most common transcendental functions. The floating point results are always rounded outwards, such that the result remains correct also under rounding errors.

There are several variants, alternatives and improvements of the approach described above (cf. [17] for a survey of the literature). These do in particular deal with stronger contractors based on non-decomposed constraints. While such could easily be included into our approach, the description in the remainder will concentrate on the simple contractors available on the decomposed form. The reasons for doing so stem from our application context: while in merely conjunctive constraint systems, non-decomposed constraints are clearly better due to their stronger contractors, completely different aspects become dominant in large and complex-structured Boolean combinations of arithmetic constraints. Here, pruning of the intervals is no longer the only forward inference mechanism, but pruning of the search space originating from the Boolean structure based on inferences from the theory side becomes at least equally important. There, decomposed constraints have their advantage, as they permit generation of more concise reasons (cf. sect. 5.3.5). It would, however, be feasible to have both the decomposed and non-decomposed forms of the constraints and their respective contractors coexist in our system, joining their virtues.

Dealing with partial operations, our implementation associates to each constraint $x = y \circ z$ in the three-address decomposition the contractor $\rho'(x) := \rho(x) \cap \rho(y) \hat{\bullet}_1 \rho(z)$ as well as all the related solved-form contractors $\rho'(y) := \rho(y) \cap \rho(x) \hat{\bullet}_2 \rho(z)$ and $\rho'(z) := \rho(z) \cap \rho(x) \hat{\bullet}_3 \rho(y)$. Together, this set of contractors is essentially equivalent to the usual contractor for primitive constraints [36], yet we take the different solved forms as being independent contractors in order to be able to trace the reasons for

contractions within conflict diagnosis. Note that each individual contraction $B' = C(e, B)$ can be decomposed into a set of individual contractions affecting just one face of $B$ each, and each having a subset of the bounds describing the faces of $B$ as a reason. E.g., in the above example, the first interval contraction derives the new bound $x \leq 3$ from the reasons $y \geq 2$ and $z \leq 5$, using equation $x + y = z$ in its solved form $x = z - y$. In the sequel, we denote such an atomic derivation of ICP by $(y \geq 2, z \leq 5) \xrightarrow{x+y=z} (x \leq 3)$.

## 5.3 The iSAT Algorithm

In order to check the satisfiability of a given formula, our algorithm essentially performs the same steps as a propositional DPLL-based SAT solver. First, it applies a definitional translation to convert the input formula into a conjunctive form, then it uses a split-and-deduce approach to solve the latter.

In section 5.3.1 we explain the definitional translation performed by our solver, which in particular involves a generalized form of the polarity-based optimization used in the pure propositional setting (see section 3.2.1). Thereafter, we present the actual search algorithm, i.e. the split-and-deduce part of iSAT. In section 5.3.2 we do so informally by means of an example, followed by an explanation of the arithmetic deduction rules employed during search in section 5.3.3. In order to enable a proof of correctness, section 5.3.4 provides a formal presentation of iSAT's split-and-deduce algorithm in form of transition-rules. Section 5.3.5 presents various optimizations of the base algorithm, and section 5.3.6 finally explains the measures taken to enforce its termination.

### 5.3.1 Definitional Translation into Conjunctive Form

Starting point for the translation is the syntax tree of the input formula whose leaves are Boolean and real-valued variables and whose inner nodes are operators.

**Definition 5.1.** An *operator* is a function of the form

$$\omega : X_1 \times \ldots \times X_n \to Y. \tag{5.1}$$

The integer $n$ (the number of *arguments* or *operands* or *inputs* the operator takes) is called the *arity* of the operator. The set $X_1 \times \ldots \times X_n$ is the *domain* of the operator, the set $Y$ is its *codomain*. We refer to 5.1 as *signature* of the operator. An operator whose

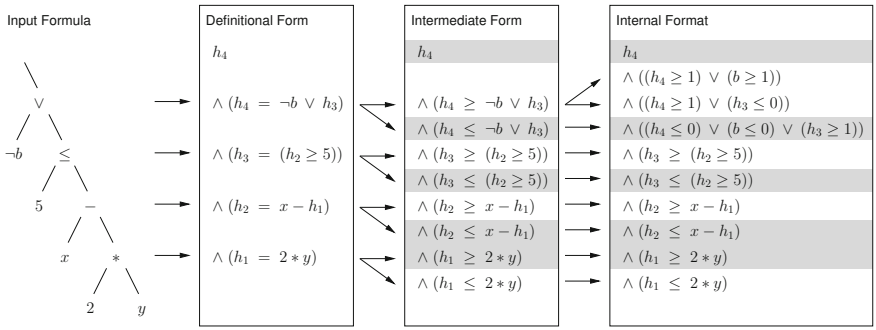


Figure 5.1: Definitional translation.

signature is $\mathbb{R}^n \to \mathbb{R}$ is called *arithmetic operator*. It is called *Boolean operator* if its signature is $\mathbb{B}^n \to \mathbb{B}$, and *relational operator* if its signature is $\mathbb{R}^2 \to \mathbb{B}$.

Given an input formula $f$, we construct, just like in the propositional case, an equisatisfiable formula by traversing the syntax tree of $f$ in preorder, replacing each subformula $g$ upon visit of its top operator node $\omega_g$ with a fresh auxiliary variable $h_g$ whose type equals the codomain of $\omega_g$, and conjoining the definition $h_g = g$ to the resulting formula $f[h_g/g]$. Note, in particular, that we treat Boolean, relational and arithmetic operators occuring in the syntax tree in exactly the same way.

**Example 5.2.** We consider the translation of $\phi = \neg b \vee (x - 2y \geq 5)$, where $b \in BV$ and $x, y \in RV$. The definitional form generated by the conversion explained above is given in the leftmost box of figure 5.1. In the course of the translation four auxiliary variables, corresponding to the four operator nodes of $\phi$, were introduced. Variables $h_1$ and $h_2$ are of type real, variables $h_3$ and $h_4$ of type Boolean.

Under certain conditions it is possible to relax the equality which constrains the auxiliary variable in a definition, to an inequality. The idea is the same as the one behind the polarity-based optimization proposed by Plaisted and Greenbaum for translation of propositional formulae (cf. sec. 3.2.1). In our setting, where we have to deal not only with Boolean definitions, but also with arithmetic ones, the notion of polarity is generalized by the concept of monotonicity. We therefore refer to the technique described in the following as *monotonicity relaxation*.

**Definition 5.3.** An operator $\omega$ with the signature $\omega : X_1 \times \ldots \times X_n \to Y$ is called *upward monotone* or *isotone* (*downward monotone* or *antitone*) in its $i$-th argument iff

for all $(a_1, \ldots, a_n) \in X_1 \times \ldots \times X_n$ the following holds: If $b_i \in X_i$ satisfies $a_i \leq b_i$ ($b_i \leq a_i$) then $\omega(a_1, \ldots, a_i, \ldots a_n) \leq \omega(a_1, \ldots, b_i, \ldots, a_n)$. If $\omega$ is neither isotone nor antitone in its $i$-th argument, we call it *non-monotone* in this argument.

**Example 5.4.**

- Negation $\neg : \mathbb{B} \to \mathbb{B}$ is antitone because $\neg 0 \geq \neg 1$.
- Implication $\to$: $\mathbb{B}^2 \to \mathbb{B}$ is antitone in its first operand because $(0 \to b) \geq (1 \to b)$ hold for all $b \in \{0, 1\}$. It is isotone in its second operand because $(b \to 0) \leq (b \to 1)$ holds for all $b \in \{0, 1\}$.
- Exclusive-Or $\nleftrightarrow$: $\mathbb{B}^2 \to \mathbb{B}$ is non-monotone in both operands. For the first operand, non-monotonicity follows from the fact that $(0 \nleftrightarrow 0) \leq (1 \nleftrightarrow 0)$, whereas $(0 \nleftrightarrow 1) > (1 \nleftrightarrow 1)$. An analogue argument is used to show non-monotonicity in the second operand.
- The greater-or-equal operator $\geq$: $\mathbb{R}^2 \to \mathbb{B}$ is isotone in its first operand because $(x \geq y) \leq (x + d \geq y)$ holds for all $x, y, d \in \mathbb{R}$ with $d \geq 0$. It is antitone in its second operand because $(x \geq y) \leq (x, y - d)$ holds for all $x, y, d \in \mathbb{R}$ with $d \geq 0$.
- Multiplication with a constant $*_c : \mathbb{R} \to \mathbb{R}$, $x \mapsto c \cdot x$, is isotone if $c \geq 0$, because then $*_c(x) \leq *_c(x + d)$ for $x, d \in \mathbb{R}$ and $d \geq 0$. Otherwise, i.e. if $c < 0$, it is antitone.

Similar arguments are used to show, for example, that conjunction $\wedge : \mathbb{B}^2 \to \mathbb{B}$, disjunction $\vee : \mathbb{B}^2 \to \mathbb{B}$, and addition $+ : \mathbb{R}^2 \to \mathbb{R}$ are isotone in both operands, whereas subtraction $- : \mathbb{R}^2 \to \mathbb{R}$ is isotone in its first operand, but antitone in its second, and multiplication $* : \mathbb{R}^2 \to \mathbb{R}$ and sinus $\sin : \mathbb{R} \to \mathbb{R}$ are non-monotone in their operands.

Being composed of nested applications of operators, a formula itself is an operator which relates the values of its subformulae to an output value. We can thus lift the notions of isotony and antitony to formulae. Let $g$ be a subformula of $f$. We say that $f$ is isotone (antitone) in $g$, if an increase of the value at the top node of $g$ causes at most an increase (a decrease) of the value at the root node of $f$, provided that all other inputs of $f$ are kept constant.

Knowing the monotonicity of the operators occuring in the syntax tree of a formula $f$, we can easily tell for a subformula $g$, whether $f$ is isotone, antitone or non-monotone in $g$. To this end we label each input edge of an operator node $\omega$ occuring in $f$ with 'isotone', 'antitone' or 'non-monotone', depending on the monotonicity of $\omega$ in the respective argument. Then the monotonicity of $f$ in a given subformula can be decided by means of the following lemma, which can be easily proven by induction over the length of the path leading from the root node of $f$ to the top node of the subformula in question.

**Lemma 5.5.** *Let $g$ be a subformula of $f$, and let $\pi$ be the path leading from the root node of $f$ to $g$. If there is an edge in $\pi$ labelled with 'non-monotone' then $f$ is non-monotone in $g$. Otherwise $f$ is isotone (antitone) in $g$ iff $\pi$ contains an even (odd) number of 'antitone'-labelled edges.*

If $f$ is monotone in the respective subformula $g$, only an inequality (instead of an equation) is needed to constrain the auxiliary variable $h_g$ in the definition introduced for $g$: If $f$ is isotone in $g$, we rewrite $f$ satisfiability-preserving to $f[h_g/g] \wedge (h_g \leq g)$. If $f$ is antitone in $g$, we rewrite it to $f[h_g/g] \wedge (h_g \geq g)$. Only if $f$ is non-monotone in $g$, we rewrite it to $f[h_g/g] \wedge (h_g = g)$.

**Example 5.6.** In the middle box of figure 5.1 the equations generated by the definitional translation have been split into pairs of inequalities. (In this example we use the relational operators $=, \leq$, and $\geq$ not only in arithmetic definitions, but also in definitions of Boolean operators, making use of the alternative notation for the Boolean connectives $\leftrightarrow, \rightarrow$, and $\leftarrow$ introduced on page 41). The highlighted inequalities are those kept by monotonicity relaxation, i.e. their conjunction is equisatisfiable to the input formula shown on the left. Note that the input formula is antitone in subformula $2 * y$, because the minus-operator is antitone in its second argument.

To complete the translation of the input formula into the internal format processed by iSAT, we rewrite Boolean definitions, just as in the standard Tseitin translation, into clauses. This is enabled by the fact that the Boolean operators $=, \leq$, and $\geq$ can be expressed in terms of conjunction, disjunction and negation, which obviously is impossible for definitions involving real-valued variables.

**Example 5.7.** The rightmost box in figure 5.1 shows the result of the final translation step. As explained in section 5.1, we use bounds on variables to represent literals that occur within clauses and write, e.g., $(b \leq 0)$ instead of $\neg b$ and $(b \geq 1)$ instead of $b$.

Actually, we not only allow bounds on Boolean variables within clauses, but also bounds on real-valued variables. By this, we can save the introduction of $h_3$ in above example and use the bound $(h_2 \geq 5)$ directly as operand of the disjunction, yielding $h_4 = (b \leq 0) \vee (h_2 \geq 5)$ as definition for the top node of the input formula. The more general format of literals is motivated by the learning scheme employed in iSAT. As we will see in the following section, explanations for conflicts learned by iSAT contain both, bounds on Boolean and on real-valued variables. In particular, the bounds on real-valued variables occuring in explanations are derived dynamically during the split-and-deduce search performed by the solver, i.e. they are not known apriori. By allowing bounds on real-valued variables to be used directly as operands of Boolean operators, we avoid the introduction of definitions (in particular of Boolean auxiliary variables) for them at runtime.

From a conceptual point of view, iSAT treats Boolean operators just in the same way as arithmetic operators. In fact, the final translation step, i.e. the conversion of Boolean definitions into clauses, is not a neccessity for the solving algorithm to work. We only perform the conversion because it enables a very simple and efficient implementation of Boolean deduction. With Boolean constraints given in clausal form, the only Boolean deduction rule needed is unit propagation. Otherwise, we would have to provide contractors for each type of Boolean definition, i.e. one for definitions of $\wedge$-nodes, one for definitions of $\vee$-nodes, and so on. This said, it might well be advantageous to retain the Boolean part of the input formula in definitional form, since the latter preserves the syntax tree structure of the formula and could thus enable the use of structural SAT approaches which directly operate on the graph structure of the input formula [99, 84, 101].[5]

Concluding this section, we remark that we again take advantage of the optimizations mentioned on page 44, adapted to the more general setting here. For example, we apply elimination of common subexpressions through re-use of the auxiliary variables, thus reducing the search space of the solver and enhancing the reasoning power of the interval contractors used in arithmetic reasoning [17].

### 5.3.2 Split-and-Deduce Search

Similar to the behaviour of a DPLL-based SAT solver, the search phase of the iSAT algorithm operates by alternating between two steps:

[5] Albeit seemingly lost by translation into clausal form, the syntax tree of the input formula can be recovered provided that information is kept whether a variable is a problem variable or an auxiliary variable introduced for conversion into clauses.

- The *decision step* involves selecting a variable, splitting its current interval (e.g. at its midpoint) and temporarily discarding either the lower or the upper half of the interval from search. The solver will ignore the discarded part of the search space until the decision is undone by backjumping.

- Each decision is followed by a *deduction step* in which the solver applies a set of deduction rules that explore all consequences of the previous decision. Essentially, these deduction rules narrow the search space by carving away portions that contain non-solutions only.

Deduction rules are applied over and over again until no further interval narrowing is achieved or the changes become negligible. Deduction may also yield a conflict, which manifests itself in that the range of a variable becomes empty, indicating the need for backjumping in order to undo decisions and their consequences. Termination of the split-and-deduce search is ensured by a) selecting a variable for splitting only if the width of its interval is above a certain threshold, and b) dropping deductions which yield negligible progress only.

It is important to understand, however, that iSAT does not manipulate intervals explicitly, but internally only deals with bounds on variables, i.e. with constraints of the form $v \sim c$, where $v \in BV \cup RV$, $\sim \in \{<, \leq, \geq, >\}$, and $c$ is a rational constant. During solving, bounds are generated dynamically by deduction and by taking decisions. The interval of a variable is, at any time, given by the strongest upper and the strongest lower bound asserted for that variable.

Before providing a formal exposition of our satisfiability solving algorithm in the following sections, we explain it by means of an example. Consider the formula

$$\begin{aligned}
\phi = \quad & ((a \leq 0) \vee (c \leq 0) \vee (d \geq 1)) && (c_1)\\
\wedge\ & ((a \leq 0) \vee (b \leq 0) \vee (c \geq 1)) && (c_2)\\
\wedge\ & ((c \leq 0) \vee (d \leq 0)) && (c_3)\\
\wedge\ & ((b \geq 1) \vee (x \geq -2)) && (c_4)\\
\wedge\ & ((x \geq 4) \vee (y \leq 0) \vee (h_3 \geq 6.2)) && (c_5)\\
\wedge\ & h_1 = x^2 && (c_6)\\
\wedge\ & h_2 = -2 \cdot y && (c_7)\\
\wedge\ & h_3 = h_1 + h_2 && (c_8)
\end{aligned}$$

which already is in the internal format processed by iSAT. Variables $a, b, c, d$ are of type Boolean, i.e. their initial intervals are $\{0, 1\}$. Variables $x, y, h_1, h_2, h_3$ are real-

valued, and we assume that their initial interval valuation is given by $\rho(x) = \rho(y) = [-10, 10]$, $\rho(h_1) = [0, 100]$, $\rho(h_2) = [-20, 20]$, and $\rho(h_3) = [-20, 120]$.

Like in DPLL-based SAT solving, iSAT maintains an implication graph which relates deductions to their reasons, both being bounds on variables. Figure 5.2 on the next page shows a series of snap-shots of the implication graph taken while solving $\phi$.

We start reasoning performing a decision and split the interval of $a$ by asserting $a \geq 1$, thus opening decision level $\mathrm{DL}_1$. Since $a \geq 1$ does not trigger any deductions, decision level $\mathrm{DL}_1$ contains only the decision itself, as shown in fig. 5.2 a). We are thus set for the next decision, which we choose to be $b \geq 1$, thereby entering $\mathrm{DL}_2$. Under the new interval assignment $\rho_1$, where $\rho_1(a) = \rho_1(b) = \{1\}$ and $\rho_1(v) = \rho(v)$ for all other variables $v$, clause $c_2$ becomes unit and propagates $c \geq 1$, which in turn triggers propagation of $d \leq 0$ by $c_3$ and, together with $a \geq 1$, of $d \leq 0$ by $c_1$, thus yielding a conflict. By cutting the implication graph as shown in fig. 5.2 b), the solver identifies the bounds $a \geq 1$ and $c \geq 1$ as explanation for the conflict and learns the conflict clause

$$((a \leq 0) \vee (c \leq 0)) \qquad (c_9)$$

in order to avoid further conflicts due to the same reason. Note that all steps described so far would be performed in the same way by a Boolean SAT solver. In fact, the scenario described in example 3.3 (see page 48) in the background section on Boolean SAT is identical to the one depicted in fig. 5.2 b).

After backtracking to $\mathrm{DL}_1$, the newly learned clause $c_9$ is unit and propagates $c \leq 0$, which in turn triggers a series of unit propagations, ending with propagation of $x \geq -2$ by $c_4$, as shown in fig. 5.2 c).

Since no further propagations are possible, the next decision is due. This time, we split the range of a real-valued variable by deciding $y \geq 4$, thereby re-entering decision level $\mathrm{DL}_2$. Given the new lower bound on $y$, we can deduce $h_2 \leq -8$ from constraint $c_7$ by ICP, see figure 5.2 d). We do not call a subordinate solver for this, but instead apply the contractor for the $*_c$-operator locally, just in the same way as we apply the unit-propagation contractor for disjunctions. Thus, we are able to combine interval constraint propagation and unit propagation in a single algorithmic framework, permitting their uniform treatment within conflict detection and conflict-driven learning.

After deciding $x \leq 3$ on decision level $\mathrm{DL}_3$, we obtain the interval valuation $\rho_2$, where in particular $\rho_2(x) = [-2, 3]$ and $\rho_2(y) = [4, 10]$ holds. Since $\rho_2 \sharp (x \geq 4)$ and also $\rho_2 \sharp (y \leq 0)$, clause $c_5$ is unit and propagates $h_3 \geq 6.2$. Furthermore, we can deduce $(x \geq -2, x \leq 3) \xrightarrow{c_6} (h_1 \leq 9)$, and thereafter $(h_1 \leq 9, h_3 \geq 6.2) \xrightarrow{c_8} (h_2 \geq -2.8)$,

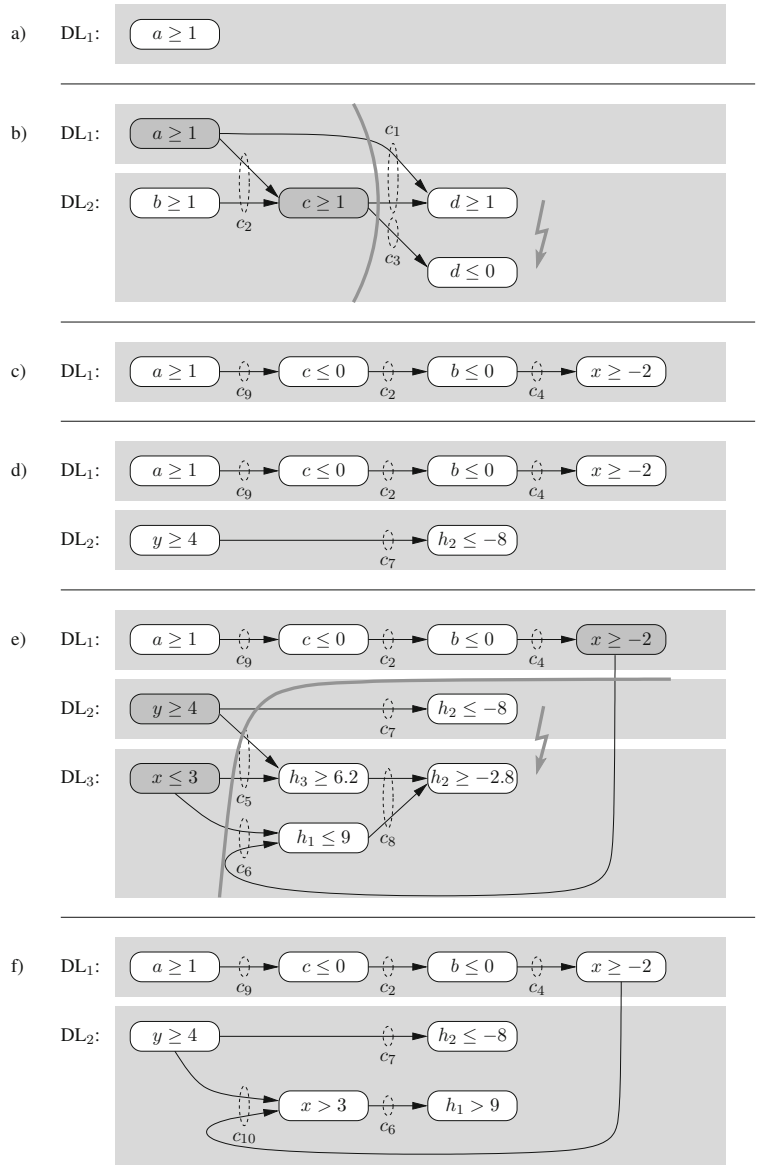


Figure 5.2: Snippet of an iSAT sample run.

this way reaching an interval valuation $\rho_3$ with $\rho_3(h_2) = \emptyset$. As in propositional SAT solving, we analyze the conflict by scanning its reasons. As illustrated in figure 5.2 e), we find, by cutting the implication graph, that bounds $x \geq -2$, $x \leq 3$, and $y \geq 4$ are a reason for the conflict. In order to avoid visiting the same branch again, we add a disjunction of the negated reason-bounds as conflict clause

$$((x < -2) \vee (x > 3) \vee (y < 4)) \qquad (c_{10})$$

to the formula and perform backjumping to decision level $DL_2$, this way resolving the conflict. We use the unique implication point technique from [125] for cutting the implication graph in order to guarantee that the conflict clause is always unit after backjumping. As illustrated in figure 5.2 f), clause $c_{10}$ propagates $x > 3$ immediately after resuming decision level $DL_2$, which in turn enables narrowing of the interval of $h_1$.

Splitting and deduction continues until either the formula turns out to be unsatisfiable or the solver is left with a 'sufficiently small' portion of the search space for which it cannot derive a conflict. Unsatisfiability is detected if a conflict is encountered, which cannot be resolved by backjumping, and 'sufficiently small' is specified by user-defined parameters controlling termination (see section 5.3.6 for details).

For equality-constraints we can only establish satisfaction if all variables occuring in such equalities have point intervals in the interval valuation. Reaching point intervals cannot be expected by naive splitting and ICP. Thus, unless unsatisfiability can be proven, iSAT outputs 'unknown' instead of 'satisfiable' in general.

### 5.3.3 Arithmetic Deduction Rules

For each type of constraint occuring in the input formula, a set of deduction rules is needed: clauses (disjunctions of bounds) are handled by unit propagation, arithmetic definitions by narrowing operators derived from interval constraint solving. In this section, we give some examples for the latter.

*Deduction Rules for Operator '+'*

The deduction rules for addition, listed in table 5.1, are used to compute inferences for definitions of the form $A = B + C$ (topmost chart) and their (by monotonicity relaxation generated) relaxed forms $A \geq B + C$ and $A \leq B + C$, respectively (charts below).

Deduction is usually performed if bounds of variables involved in a definition have changed. Assume that $\rho(A) = [a_1, a_2] = [-4, 0]$, $\rho(B) = [b_1, b_2] = [1, 4]$, and

a)

| | Deduced Bounds | Reason |
|---|---|---|
| $A = B + C$ | $a_1 = b_1 + c_1$ | $\{b_1, c_1\}$ |
| | $a_2 = b_2 + c_2$ | $\{b_2, c_2\}$ |
| $B = A - C$ | $b_1 = a_1 - c_2$ | $\{a_1, c_2\}$ |
| | $b_2 = a_2 - c_1$ | $\{a_2, c_1\}$ |
| $C = A - B$ | $c_1 = a_1 - b_2$ | $\{a_1, b_2\}$ |
| | $c_2 = a_2 - b_1$ | $\{a_2, b_1\}$ |

b)

| | Deduced Bounds | Reason |
|---|---|---|
| $A \geq B + C$ | $a_1 = b_1 + c_1$ | $\{b_1, c_1\}$ |
| $B \leq A - C$ | $b_2 = a_2 - c_1$ | $\{a_2, c_1\}$ |
| $C \leq A - B$ | $c_2 = a_2 - b_1$ | $\{a_2, b_1\}$ |

c)

| | Deduced Bounds | Reason |
|---|---|---|
| $A \leq B + C$ | $a_2 = b_2 + c_2$ | $\{b_2, c_2\}$ |
| $B \geq A - C$ | $b_1 = a_1 - c_2$ | $\{a_1, c_2\}$ |
| $C \geq A - B$ | $c_1 = a_1 - b_2$ | $\{a_1, b_2\}$ |

Table 5.1: Deduction rules for addition.

$\rho(C) = [c_1, c_2] = [-3, 5]$ holds when deduction for the triplet $A = B + C$ starts. First, we apply the rules given in the top row of chart a) in order to potentially derive new bounds for $A$. Indeed, $a_1 = b_1 + c_1 = 1 + (-3) = -2$ gives us a lower bound of $-2$ on $A$ which is tighter than the previously known bound of $-4$. We thus assert the newly deduced bound $A \geq -2$ and record bounds $b_1$ and $c_1$ as reasons for it. Hence, $B \geq 1$ and $C \geq -3$ will be antecedent nodes of $A \geq -2$ in the implication graph. The bounds to be used as reason are, for each deduction rule, given in the rightmost column of the chart. Application of rule $a_2 = b_2 + c_2$ yields $9$ as upper bound of $A$, which, however, is weaker than the current upper bound $A \leq 0$, and is therefore dropped.

Applying the remaining rules from chart a), we obtain $b_2 = a_2 - c_1 = 0 - (-3) = 3$ as new upper bound for $B$, and $c_2 = a_2 - b_1 = 0 - 1 = -1$ a new upper bound for $C$. The lower bounds $B$ and $C$ are weaker than the ones already in place and therefore ignored. Thus, the final interval valuation after deduction is $\hat{\rho}(A) = [-2, 0]$, $\hat{\rho}(B) = [1, 3]$, and $\hat{\rho}(C) = [-3, -1]$.

Note that, in general, the deduction routine of our solver does not apply *all* rules provided in table 5.1 a), but only those where the bound which triggered deduction

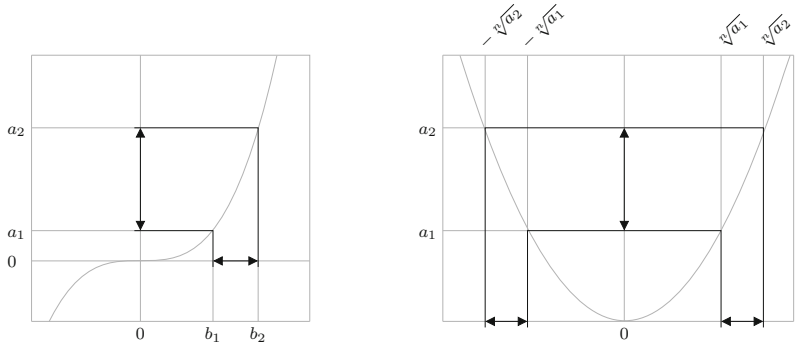


Figure 5.3: Images and pre-images of intervals under the mapping $A = B^n$ for odd exponents (left) and for even exponents (right).

| | Deduced Bounds | Reason |
|---|---|---|
| $A = B^n$ | $a_1 = b_1^n$ | $\{b_1\}$ |
| | $a_2 = b_2^n$ | $\{b_2\}$ |
| $B = \sqrt[n]{A}$ | $b_1 = \sqrt[n]{a_1}$ | $\{a_1\}$ |
| | $b_2 = \sqrt[n]{a_2}$ | $\{a_2\}$ |

Table 5.2: Deduction rules for exponentiation with odd exponent $n \in \mathbb{N}$.

occurs on the right-hand side. The newly computed bounds may then trigger further visits of the same triplet, entailing more deductions.

Obviously, an inequality of form $A \geq B + C$ allows only the deduction of a lower bound for $A$ and of upper bounds for $B$ and $C$. For inequalities of form $A \leq B + C$ the situation is just reversed. The respective deduction rules are given in charts b) and c) of table 5.1.

So far we have treated all bounds as if they were non-strict. Still, we have to take strictness of bounds into account during deduction. As a rule of thumb one can say, that a deduced bound is strict if a strict bound was involved in its computation.

*Deduction Rules for Operator '$x^n$'*

Due to the monotonicity of functions of type $A = B^n$ for odd exponents, the deduction rules for exponentiation are very simple in this case (see table 5.2). For even exponents, the graph of $A = B^n$ is non-monotone, however, which in particular means

| | Condition | Deduced Bounds | Reason |
|---|---|---|---|
| $A = B^n$ | $(b_1 \geq 0)$ | $a_1 = b_1^n$ | $\{b_1\}$ |
| | | $a_2 = b_2^n$ | $\{b_1, b_2\}$ |
| | $(b_2 \leq 0)$ | $a_1 = b_2^n$ | $\{b_2\}$ |
| | | $a_2 = b_1^n$ | $\{b_1, b_2\}$ |
| | $(b_1 < 0 < b_2)$ | $a_1 = 0$ | $\{b_2\}$ |
| | | $a_2 = \max(b_1^n, b_2^n)$ | $\{b_1, b_2\}$ |
| $B = \sqrt[n]{A}$ | $(b_2 > \sqrt[n]{a_2})$ | $b_2 = \sqrt[n]{a_2}$ | $\{a_2, b_2\}$ |
| | $(b_2 < \sqrt[n]{a_1})$ | $b_2 = -\sqrt[n]{a_1}$ | $\{a_1, b_2\}$ |
| | $(b_1 < -\sqrt[n]{a_2})$ | $b_1 = -\sqrt[n]{a_2}$ | $\{a_2, b_1\}$ |
| | $(b_1 > -\sqrt[n]{a_1})$ | $b_1 = \sqrt[n]{a_1}$ | $\{a_1, b_1\}$ |

Table 5.3: Deduction rules for exponentiation with even exponent $n \in \mathbb{N}$.

that an interval of $A$ can have two pre-images on the $B$-axis, as illustrated in figure 5.3. To enable optimal pruning of intervals, we have to resort to case splitting, i.e. the deduction rules for even exponents come with application conditions, as can be seen in table 5.3. We illustrate their use by means of an example. Assume that $n = 2$, $\rho(A) = [4, 16]$, and $\rho(B) = [-3, 1]$. Because $b_1 = -3 < 0 < 1 = b_2$, we can deduce $a_1 = 0$ and $a_2 = \max((-3)^2, 1^2) = 9$, contracting the interval of $A$ to $\hat{\rho}(A) = [4, 9]$. $A$'s new interval has the pre-images $[2, 3]$ and $[-3, -2]$. But since the upper bound of $B$ is stricly less than $\sqrt[2]{4}$, i.e. condition $b_2 < \sqrt[n]{a_1}$ applies, we can exclude pre-image [2, 3] and deduce $b_2 = -\sqrt[n]{a_1} = -\sqrt[2]{4} = -2$ as new upper bound of $B$. The final interval valuation is thus $\hat{\rho}(A) = [4, 9]$ and $\hat{\rho}(B) = [-3, -2]$.

An important point to note is that $b_2$ is part of the reason of the last deduction, although it does not occur on the right-hand side of the deduction rule $b_2 = -\sqrt[n]{a_1}$. This is, because $b_2$ occurs in the *application condition* of the deduction rule. Failing to include $b_2$ in the reason (and thus in the implication graph) would be an error, because it may entail the derivation of conflict clauses which prune the search space too aggressively, cutting off not only non-solutions but also potential solutions.

To simplify the presentation of the deduction rules, we have again treated all bounds as if they were non-strict. This allowed us to express bounds, which are in fact inequalities relating a variable to a constant, in form of a constant only. Of course, an implementation of the rules has to take strictness into account. In particular, the application conditions given in table 5.3 have to be interpreted correspondingly when strictness comes into play. For example, condition $b_2 < \sqrt[n]{a_1}$ states that the upper bound $b_2$ and the lower bound $\sqrt[n]{a_1}$ together define an empty interval. Taking into ac-

count strictness, the bounds involved in the condition are objects of the form $B \sim_1 b_2$ and $B \sim_2 \sqrt[n]{a_1}$, where $\sim_1 \in \{<, \leq\}$ and $\sim_2 \in \{>, \geq\}$. Consequently, $b_2 < \sqrt[n]{a_1}$ translates to unsatisfiability of

$$(B \sim_1 b_2) \wedge (B \sim_2 \sqrt[n]{a_1}).$$

Similarly, the application condition $b_1 < -\sqrt[n]{a_2}$, stating that the lower bound $-\sqrt[n]{a_2}$ is stronger than the lower bound $b_1$, has to be interpreted as implication

$$(B \sim_1 b_1) \leftarrow (B \sim_2 -\sqrt[n]{a_2}),$$

where $\sim_1, \sim_2 \in \{>, \geq\}$, if strictness is considered. For sake of clarity, we will continue to neglect the issue of strictness in the presentation of the remaining deduction rules, and tacitly assume that rules and conditions are understood as if they were formulated for general (potentially strict) bounds.

*Deduction Rules for Operator '$*$'*

Table 5.4 on the facing page provides the deduction rules for multiplication of two variables, i.e. for triplets of form $A = B \cdot C$. The set of rules includes, like those presented before, in particular deduction rules for the inverse operations, i.e. for $B = A/C$ and $C = A/B$, in order to enable narrowing of *all* variables occuring in the triplet. Hence, we can use the same rules to implement a contractor for division[6], and, similarly, implement deduction for subtraction and extraction of roots using the rules for addition and exponentiation.

*Deduction Rules for the Cosine Operator*

When generating a definition of type $A = \cos(B)$ in the solver frontend, we immediately constrain the range of $A$ to the interval $[-1, 1]$ by adding $(A \geq -1) \wedge (A \leq 1)$ to the formula. We cannot contract $A$'s interval any further unless the width of $B$'s interval $[b_1, b_2]$ falls below $2\pi$. Hence, this is checked first when entering the deduction routine for cosine because of changes of $b_1$ or $b_2$. If $b_2 - b_1 < 2\pi$, we shift the interval $[b_1, b_2]$ such that the left interval border lies within $[0, 2\pi]$. We then apply the deduction rules given in table 5.5 a) to the shifted interval $[b'_1, b'_2]$, whose borders are

$$b'_1 = b_1 - 2\pi \cdot \left\lfloor \frac{b_1}{2\pi} \right\rfloor \tag{5.2}$$

$$b'_2 = b_2 - 2\pi \cdot \left\lfloor \frac{b_1}{2\pi} \right\rfloor, \tag{5.3}$$

[6] Alternatively, we could rewrite definitions of form $A = B/C$ into $B = A \cdot C \wedge C \neq 0$ in the frontend of the solver.

| | Condition | Deduced Bounds | Reason |
|---|---|---|---|
| $A = B \cdot C$ | $(b_1 \geq 0) \wedge (c_1 \geq 0)$ | $a_1 = b_1 \cdot c_1$ | $\{b_1, c_1\}$ |
| | | $a_2 = b_2 \cdot c_2$ | $\{b_1, b_2, c_1, c_2\}$ |
| | $(b_1 \geq 0) \wedge (c_2 \leq 0)$ | $a_1 = b_2 \cdot c_1$ | $\{b_1, b_2, c_1, c_2\}$ |
| | | $a_2 = b_1 \cdot c_2$ | $\{b_1, c_2\}$ |
| | $(b_1 \geq 0) \wedge (c_1 < 0 < c_2)$ | $a_1 = b_2 \cdot c_1$ | $\{b_1, b_2, c_1, c_2\}$ |
| | | $a_2 = b_2 \cdot c_2$ | $\{b_1, b_2, c_1, c_2\}$ |
| | $(b_2 \leq 0) \wedge (c_1 \geq 0)$ | $a_1 = b_1 \cdot c_2$ | $\{b_1, b_2, c_1, c_2\}$ |
| | | $a_2 = b_2 \cdot c_1$ | $\{b_2, c_1\}$ |
| | $(b_2 \leq 0) \wedge (c_2 \leq 0)$ | $a_1 = b_2 \cdot c_2$ | $\{b_2, c_2\}$ |
| | | $a_2 = b_1 \cdot c_1$ | $\{b_1, b_2, c_1, c_2\}$ |
| | $(b_2 \leq 0) \wedge (c_1 < 0 < c_2)$ | $a_1 = b_1 \cdot c_2$ | $\{b_1, b_2, c_1, c_2\}$ |
| | | $a_2 = b_1 \cdot c_1$ | $\{b_1, b_2, c_1, c_2\}$ |
| | $(b_1 < 0 < b_2) \wedge (c_1 \geq 0)$ | $a_1 = b_1 \cdot c_2$ | $\{b_1, b_2, c_1, c_2\}$ |
| | | $a_2 = b_2 \cdot c_2$ | $\{b_1, b_2, c_1, c_2\}$ |
| | $(b_1 < 0 < b_2) \wedge (c_2 \leq 0)$ | $a_1 = b_2 \cdot c_1$ | $\{b_1, b_2, c_1, c_2\}$ |
| | | $a_2 = b_1 \cdot c_1$ | $\{b_1, b_2, c_1, c_2\}$ |
| | $(b_1 < 0 < b_2) \wedge (c_1 < 0 < c_2)$ | $a_1 = \min(b_1 \cdot c_2, b_2 \cdot c_1)$ | $\{b_1, b_2, c_1, c_2\}$ |
| | | $a_2 = \max(b_1 \cdot c_1, b_2 \cdot c_2)$ | $\{b_1, b_2, c_1, c_2\}$ |
| $C = A/B$ | $(a_1 \geq 0) \wedge (b_1 > 0)$ | $c_1 = a_1/b_2$ | $\{b_1, b_2, a_1\}$ |
| | | $c_2 = a_2/b_1$ | $\{b_1, a_1, a_2\}$ |
| | $(a_1 \geq 0) \wedge (b_2 < 0)$ | $c_1 = a_2/b_2$ | $\{b_2, a_1, a_2\}$ |
| | | $c_2 = a_1/b_1$ | $\{b_1, b_2, a_1\}$ |
| | $(a_2 \leq 0) \wedge (b_1 > 0)$ | $c_1 = a_1/b_1$ | $\{b_1, a_1, a_2\}$ |
| | | $c_2 = a_2/b_2$ | $\{b_1, b_2, a_2\}$ |
| | $(a_2 \leq 0) \wedge (b_2 < 0)$ | $c_1 = a_2/b_1$ | $\{b_1, b_2, a_2\}$ |
| | | $c_2 = a_1/b_2$ | $\{b_2, a_1, a_2\}$ |
| | $(a_1 < 0 < a_2) \wedge (b_1 > 0)$ | $c_1 = a_1/b_1$ | $\{b_1, a_1, a_2\}$ |
| | | $c_2 = a_2/b_1$ | $\{b_1, a_1, a_2\}$ |
| | $(a_1 < 0 < a_2) \wedge (b_2 < 0)$ | $c_1 = a_2/b_2$ | $\{b_2, a_1, a_2\}$ |
| | | $c_2 = a_1/b_2$ | $\{b_2, a_1, a_2\}$ |
| $B = A/C$ | $(a_1 \geq 0) \wedge (c_1 > 0)$ | $b_1 = a_1/c_2$ | $\{c_1, c_2, a_1\}$ |
| | | $b_2 = a_2/c_1$ | $\{c_1, a_1, a_2\}$ |
| | $(a_1 \geq 0) \wedge (c_2 < 0)$ | $b_1 = a_2/c_2$ | $\{c_2, a_1, a_2\}$ |
| | | $b_2 = a_1/c_1$ | $\{c_1, c_2, a_1\}$ |
| | $(a_2 \leq 0) \wedge (c_1 > 0)$ | $b_1 = a_1/c_1$ | $\{c_1, a_1, a_2\}$ |
| | | $b_2 = a_2/c_2$ | $\{c_1, c_2, a_2\}$ |
| | $(a_2 \leq 0) \wedge (c_2 < 0)$ | $b_1 = a_2/c_1$ | $\{c_1, c_2, a_2\}$ |
| | | $b_2 = a_1/c_2$ | $\{c_2, a_1, a_2\}$ |
| | $(a_1 < 0 < a_2) \wedge (c_1 > 0)$ | $b_1 = a_1/c_1$ | $\{c_1, a_1, a_2\}$ |
| | | $b_2 = a_2/c_1$ | $\{c_1, a_1, a_2\}$ |
| | $(a_1 < 0 < a_2) \wedge (c_2 < 0)$ | $b_1 = a_2/c_2$ | $\{c_2, a_1, a_2\}$ |
| | | $b_2 = a_1/c_2$ | $\{c_2, a_1, a_2\}$ |

Table 5.4: Deduction rules for multiplication.

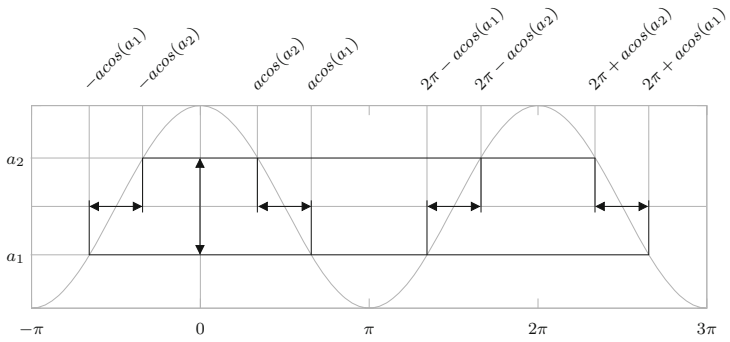


Figure 5.4: The interval $[a_1, a_2]$ has four pre-images in $[-\pi, 3\pi]$.

in order to (potentially) obtain tightened bounds for variable $A$. The rules needed to narrow the interval of $B$ based on an interval of $A$ are listed in table 5.5 b). Figure 5.4 illustrates the constants used in the rules. Before the rules can be applied, the current bounds of $B$ have to be shifted into the interval $[0, 2\pi]$, i.e. application conditions and deduction rules are evaluated w.r.t. the shifted bounds

$$b'_1 = b_1 - 2\pi \cdot \left\lfloor \frac{b_1}{2\pi} \right\rfloor \tag{5.4}$$

$$b'_2 = b_2 - 2\pi \cdot \left\lfloor \frac{b_2}{2\pi} \right\rfloor . \tag{5.5}$$

Note that (5.2) and (5.3) shift upper and lower bound of $B$ by the same amount, whereas (5.4) and (5.5) shift the bounds separately. Of course, the effect of shifting has to be undone before asserting the tightened bounds determined by deduction.

### 5.3.4 Correctness

The underlying idea of iSAT is that the two central operations of ICP-based arithmetic constraint solving — interval contraction by constraint propagation and by interval splitting — correspond to asserting bounds on real-valued variables $v \sim c$ with $v \in RV$, $\sim \in \{<, \leq, \geq, >\}$ and $c \in \mathbb{Q}$. Likewise, the decision steps and unit propagations in DPLL proof search correspond to asserting literals. A unified DPLL- and ICP-based proof search on a formula $\phi$ from the formula language of section 5.1 can thus be based on asserting or retracting bounds, thereby in lockstep refining or widening an interval valuation $\rho$ that represents the current set of candidate solutions.

a)

| | Condition | Deduced Bounds | Reason |
|---|---|---|---|
| $A = \cos(B)$ | $(0 \leq b_1 \leq \pi) \wedge (0 \leq b_2 \leq \pi)$ | $a_1 = \cos(b_2)$ | $\{b_1, b_2\}$ |
| | | $a_2 = \cos(b_1)$ | $\{b_1, b_2\}$ |
| | $(\pi \leq b_1 \leq 2\pi) \wedge (\pi \leq b_2 \leq 2\pi)$ | $a_1 = \cos(b_1)$ | $\{b_1, b_2\}$ |
| | | $a_2 = \cos(b_2)$ | $\{b_1, b_2\}$ |
| | $(0 \leq b_1 \leq \pi) \wedge (\pi \leq b_2 \leq 2\pi)$ | $a_2 = \max(\cos(b_1), \cos(b_2))$ | $\{b_1, b_2\}$ |
| | $(\pi \leq b_1 \leq 2\pi) \wedge (2\pi \leq b_2 \leq 3\pi)$ | $a_1 = \min(\cos(b_1), \cos(b_2))$ | $\{b_1, b_2\}$ |

b)

| | Condition | Deduced Bounds | Reason |
|---|---|---|---|
| $B = \operatorname{acos}(A)$ | $2\pi - \operatorname{acos}(a_2) < b_1$ | $b_1 = 2\pi + \operatorname{acos}(a_2)$ | $\{a_2, b_1\}$ |
| | $\operatorname{acos}(a_1) < b_1 < 2\pi - \operatorname{acos}(a_1)$ | $b_1 = 2\pi - \operatorname{acos}(a_1)$ | $\{a_1, b_1\}$ |
| | $b_1 < \operatorname{acos}(a_2)$ | $b_1 = \operatorname{acos}(a_2)$ | $\{b_1, a_2\}$ |
| | $2\pi - \operatorname{acos}(a_2) < b_2$ | $b_2 = 2\pi - \operatorname{acos}(a_2)$ | $\{a_2, b_2\}$ |
| | $\operatorname{acos}(a_1) < b_2 < 2\pi - \operatorname{acos}(a_1)$ | $b_2 = \operatorname{acos}(a_1)$ | $\{a_1, b_2\}$ |
| | $b_2 < \operatorname{acos}(a_2)$ | $b_2 = -\operatorname{acos}(a_2)$ | $\{a_2, b_2\}$ |

Table 5.5: Deduction rules for cosine operation.

In our algorithm we use the letter $M$ to denote the list of asserted bounds. In order to allow backtracking on this data-structure, in addition, we intersperse a special marker symbol $|$ into this list $M$. The asserted bounds comprised in $M$ induce an interval assigment $\rho_M$ which assigns to each variable $v \in BV \cup RV$ an interval which is defined by the tightest bounds on $v$ occuring in $M$, i.e., $\rho_M(v) := \bigcap_{(v \sim c) \in M} \{u \mid u \in \mathbb{R}, u \sim c\}$.

The algorithm maintains a 2-tuple $(M, \phi)$ as its proof state, where $M$ is the list of asserted bounds, and $\phi$ a formula. For the basic procedure, $\phi$ will remain constant and always be equal to the formula to be solved. It is not before introducing conflict-driven learning that we will see changes to $\phi$. The procedure searching for satisfying valuations then proceeds as follows:

*Step 1: Initialization*

Proof search on the input formula $\phi$ starts from the inital state $(M_0, \phi)$, where $M_0$ contains the bounds defining the inital ranges of the variables, i.e. $\{(v \geq 0), (v \leq 1)\} \subseteq M_0$ if $v \in BV$, and $\{(v \geq l), (v \leq u)\} \subseteq M_0$ if $v \in RV$ is a problem variable with range $[l, u]$.

*Step 2: Deduction*

Proof search continues with searching for *unit clauses* in $\phi$, that is, clauses that have all but one atoms being inconsistent with the current interval valuation $\rho_M$.

If such a clause is found then the remaining bound is asserted:

$$\frac{\begin{array}{c}\phi = \phi' \wedge (a_1 \vee \ldots \vee a_n),\\ \exists j \in \mathbb{N}_{\leq n} \,.\, \forall i \in \mathbb{N}_{\leq n} \,.\, (i \neq j \;\Rightarrow\; (\rho \,\sharp\, a_i)), \rho_M \not\models_{\mathrm{hc}} a_j\end{array}}{(M, \phi) \longrightarrow (M \cdot \langle a_j \rangle, \phi)} \tag{5.6}$$

Likewise, contractions obtained from *definitions* (by application of the deduction rules explained in section 5.3.3) are asserted as bounds:

$$\frac{\begin{array}{c}\phi = \phi' \;\wedge\; e,\\ \exists\, b_1, \ldots, b_n \in M \,.\, (b_1, \ldots, b_n) \xrightarrow{e} (v \sim c)\,, \rho_M \not\models_{\mathrm{hc}} v \sim c\end{array}}{(M, \phi) \longrightarrow (M \cdot \langle v \sim c \rangle, \phi)} \tag{5.7}$$

where $e$ is a definition and the $b_i$ are bounds. Note that rule (5.7) is different in spirit from theory-related rules in DPLL($T$), as it does neither analyze consistency of the currently asserted set of theory atoms nor implicative relations between these and other theory atoms occurring in the input formula (as in DPLL($T$)). Instead, it applies purely local reasoning with respect to the single theory atom $e$ and the bounds (i.e., domain restrictions) $b_1$ to $b_n$, generating a fresh bound $v \sim c$ not present in the original formula. Consistency is never tested on the full set of (nonlinear) theory atoms, but only within the extremely simple sub-theory of bound atoms through rules (5.9) and (5.10).

Step 2 is repeated until contraction detects a conflict in the sense of some interval $\rho_M(v)$ becoming empty, which is handled by continuing at step 4, or until no further contraction is obtained.[7]

*Step 3: Splitting*

In the latter case, iSAT applies a *splitting step*: it selects a variable $v \in BV \cup RV$ that is interpreted by a non-point interval and splits its interval $\rho_M(v)$ by asserting a bound that contains $v$ as a free variable and which is inconclusive on $\rho_M$. Note that the complement of such an assertion also is a bound and is inconclusive on $\rho_M$ too. In our current implementation, we use the usual strategy of bisection, i.e., the choice of $c$ as the midpoint of $\rho_M(v)$.

$$\frac{v \sim c \text{ inconclusive on } \rho_M, v \text{ occurs in } \phi}{(M, \phi) \longrightarrow (M \cdot \langle |, v \sim c \rangle, \phi)} \tag{5.8}$$

[7] In practice, one stops as soon as the changes become negligible.

Note that we use the marker symbol | to indicate the position where the new decision level starts within $M$. After the split, the algorithm continues at step 2.

*Step 4: Handling of Conflicts*

In case of a conflict, some previous splits (cf. step 3) have to be reverted, which is achieved by backtracking — thereby undoing all assertions being consequences of the split — and by asserting the complement of the previous split. In $M$, the split is marked by the special symbol | preceding the atom asserted by the split.

$$\frac{\rho'_M(w) = \emptyset \text{ for some } w \in BV \cup RV, \, | \notin M'}{(M \cdot \langle |, v \sim c \rangle \cdot M', \phi) \longrightarrow (M \cdot \langle v \not\sim c \rangle, \phi)} \tag{5.9}$$

Later (in section 5.3.5 below) we will see that, beyond backtracking, information about the reason for the conflict can be recorded in the formula $\phi$, thus pruning the remaining search space. If rule (5.9) is applicable, i.e. if there is an open backtracking point marked by the split marker '|' in the list of asserted atoms, then we apply the rule and proceed to step 2. Otherwise, i.e. if there is no previous split with a yet unexplored alternative, then the algorithm stops with result 'unsatisfiable'.

$$\frac{\rho_M(w) = \emptyset \text{ for some } w \in BV \cup RV, \, | \notin M}{(M, \phi) \longrightarrow \text{unsat}} \tag{5.10}$$

The correctness of the algorithm rests on the following two invariance properties preserved by rules (5.6) to (5.9):

**Lemma 5.8.** *Assume* $(M_0, \phi) \longrightarrow^* (M, \phi) \longrightarrow (M', \phi)$. *Then*

1. $\eta \models_{\text{hc}} \left(\bigwedge_{a \in M} a\right)$
   *implies that* $\eta$ *is a (not necessarily proper) refinement of* $\rho_M$,

2. $\text{hcsat}\left(\phi \wedge \left(\bigwedge_{a \in M} a \vee \neg \bigwedge_{b \in C_M} b\right)\right)$
   *implies* $\text{hcsat}\left(\phi \wedge \left(\bigwedge_{a' \in M'} a' \vee \neg \bigwedge_{b' \in C_{M'}} b'\right)\right)$

*hold, where* $C_N = \{x \sim c \mid N = N_1 \cdot \langle |, x \sim c \rangle \cdot N_2\}$ *is the set of choice points in* $N$.

*Proof.* Property 1 follows directly from the definitions, since $\eta \models_{\mathrm{hc}} (\bigwedge_{a\in M} a)$

$$\begin{aligned}
&\text{iff } \eta \models_{\mathrm{hc}} a \text{ for all } a \in M\\
&\text{iff } \eta \models_{\mathrm{hc}} (v \sim c) \text{ for all } (v \sim c) \in M\\
&\text{iff } \eta(v) \subseteq \{u \mid u \in \mathbb{R}, u \sim c\} \text{ for all } (v \sim c) \in M\\
&\text{iff } \eta(v) \subseteq \bigcap_{(v\sim c)\in M} \{u \mid u \in \mathbb{R}, u \sim c\} = \rho_M(v) \text{ for all variables } v.
\end{aligned}$$

Thus, $\eta(v) \subseteq \rho_M(v)$ for all variables $v$, that is, $\eta$ is a refinement of $\rho_M$.

Property 2 requires a case analysis w.r.t. the changes applied to $M$: Within rules (5.6) and (5.7), $M$ is expanded by deduced atoms $a'$ that the different contractors (unit propagation, interval contraction) permit to be drawn from $\phi \wedge \bigwedge_{a\in M} a$. Each such atom $a'$ satisfies the contractor soundness condition $\eta \models_{\mathrm{hc}} \phi \wedge \bigwedge_{a\in M} a$ iff $\eta \models_{\mathrm{hc}} \phi \wedge \bigwedge_{a\in M} a \wedge a'$ for each refinement $\eta$ of $\rho_M$. As property 1 shows that $\phi \wedge \bigwedge_{a\in M} a$ can only be satisfied by refinements of $\rho_M$, the conjectured implication follows for rules (5.6) and (5.7) from the fact that $M' = M \cup \{a'\}$ and $C_{M'} = C_M$. The splitting rule (5.8) adds a split bound $x \sim c$ which occurs in both $M'$ and $C_{M'}$ such that the conjectured implication holds due to absorption. For rule (5.9) we observe that due to the premise $\rho_M(w) = \emptyset$ of the rule, property 1 of the Lemma gives $\eta(w) = \emptyset$ for each interval valuation $\eta$ with $\eta \models_{\mathrm{hc}} \bigwedge_{a\in M} a$. That is, $\bigwedge_{a\in M} a$ is not hull consistently satisfiable. Consequently, either $\phi \wedge \big(\bigwedge_{a\in M} a \vee \neg \bigwedge_{b\in C_M} b\big)$ is not hull consistently satisfiable (in which case the implication trivially follows), or $\phi \wedge \neg \bigwedge_{b\in C_M} b$ is hull consistently satisfiable. The latter implies hull consistent satisfiability of $\phi \wedge \Big(\bigwedge_{a'\in M'} a' \vee \neg \bigwedge_{b'\in C_{M'}} b'\Big)$, as $C_{M'} = C_M \setminus \{v \sim c\}$ and $v \not\sim c \in M'$ for some bound $v \sim c$. Rule (5.10), finally, does not yield a configuration of the form $(M', \phi)$ such that it trivially satisfies the conjecture. □

**Corollary 5.9.** *If* $(M_0, \phi) \longrightarrow^* (M, \phi) \longrightarrow \mathrm{unsat}$ *then* $\phi \wedge \bigwedge_{a\in M_0}$*, i.e.* $\phi$ *with its variables confined to their initial ranges, is unsatisfiable.*

*Proof.* We assume that $(M_0, \phi) \longrightarrow^* (M, \phi) \longrightarrow \mathrm{unsat}$. Then, according to the premises of rule (5.10), $\rho_M(w) = \emptyset$ for some $w \in BV \cup RV$ and, furthermore, $| \notin M$. By induction over the length of the derivation sequence and Lemma 5.8, property 2, we obtain that $\mathrm{hcsat}\big(\phi \wedge \bigwedge_{a\in M_0} a\big)$ implies $\mathrm{hcsat}\big(\phi \wedge \big(\bigwedge_{a\in M} a \vee \neg \bigwedge_{b\in C_M} b\big)\big)$. Because $| \notin M$ and thus $C_M = \emptyset$, this in turn gives: $\mathrm{hcsat}\big(\phi \wedge \bigwedge_{a\in M_0} a\big)$ implies $\mathrm{hcsat}\big(\phi \wedge \bigwedge_{a\in M} a\big)$. According to Lemma 5.8, property 1, any interval valuation $\eta$ with $\eta \models_{\mathrm{hc}} \phi \wedge \bigwedge_{a\in M} a$ is a refinement of $\rho_M$, i.e. has $\eta(w) = \emptyset$. Thus, $\phi \wedge \bigwedge_{a\in M} a$ is

not hull consistently satisfiable, which implies that $\phi \wedge \bigwedge_{a \in M_0}$ is not hull consistently satisfiable. As hull consistent satisfiability is a necessary condition for satisfiability over the reals, it follows that $\phi \wedge \bigwedge_{a \in M_0}$ is unsatisfiable. □

### 5.3.5 Algorithmic Enhancements

By its similarity to DPLL algorithms, this base algorithms lends itself to all the algorithmic enhancements and sophisticated data structures that were instrumental to the impressive recent gains in propositional SAT solver performance.

*Lazy Clause Evaluation*

In order to save costly visits to and evaluations of disjunctive clauses, we extend the lazy clause evaluation scheme of zChaff (cf. sec. 3.2.2, p. 49) to our more general class of atoms: within each clause, we select two bounds which are inconclusive w.r.t. the current valuation $\rho$, called the 'watched bounds' of the clause. Instead of scanning the whole clause set for unit clauses in step 2 of the base algorithm, we only visit a clause if one of its two watched bounds might be inconsistent with the bound which has been asserted last. In this case, we evaluate the bounds's truth value. If found to be inconsistent w.r.t. the new interval assignment, the algorithm tries to substitute the bound by a currently unwatched and not yet inconsistent bound to watch in the future. If this substitution fails due to all remaining bounds in the clause being inconsistent, the clause has become unit and the second watched bound has to be propagated using rule (5.6).

*Watched Bounds in Arithmetic Definitions*

To save dispensable calls to contractors for arithmetic definitions (i.e. for triplets and pairs), we apply the concept of 'watched bounds' also to the latter. Consider, for example, the definition

$$x \leq y - z \tag{5.11}$$

which, due to the direction of its relational operator, may become violated if either $x$ increases or $y - z$ decreases, i.e. if $y$ decreases or $z$ increases. Thus, satisfiability of inequality (5.11) is endangered if the upper bound of $y$ or the lower bounds of $x$ and $z$ are tightened, which we watch therefore. Only if one of the watched bounds changes, we have to evaluate the triplet (i.e., we have to call the corresponding contractor) in order to be able to perform the required deductions.

*Maintaining an Implication Graph*

In order to be able to tell reasons for conflicts (i.e., empty interval valuations) encountered, our solver maintains an implication graph $IG$ akin to that known from propositional SAT solving [125]. As all asserted bounds are recorded in the stack-like data structure $M$, the implication graph is implemented by way of pointers providing backward references within $M$. Each asserted bound in $M$ then has a set of pointers naming the reasons (if any) for its assertion. That is, after application of rule (5.6), the entry $a_j$ in $M$ is decorated with pointers to the *reasons* for the entries $a_i$ with $i \neq j$ being inconsistent. These reasons are bounds already asserted in $M$: if $a_i$ in rule (5.6) is a bound $v \sim c$ then $M$ contains another bound $v \sim' c'$ with $v \sim c \wedge v \sim' c'$ being unsatisfiable, in which case $v \sim' c'$ can serve as a reason. When applying rule (5.7), the reasons are apparent from the contraction enforced, as explained in section 5.3.3: in the contraction $(b_1, \ldots, b_n) \xrightarrow{e} (v \sim c)$, $b_1, \ldots, b_n$ are the reasons for the bound $v \sim c$. The other rules do not record reasons because they either do not assert atoms (5.9) or the asserted atoms originate in choices (rules 5.8), which is recorded by attaching an empty set of reasons to the asserted bound. Note how the homogeneous treatment of Boolean and theory-related reasoning in our framework simplifies extraction of the implication graph: as both Boolean constraint propagations and theory-related constraint propagations are bound assignments in the same list $M$ of asserted bounds, rather than deferring the theory reasoning to a subordinate theory solver checking theory consistency, the implication graph can be maintained via links in $M$ only.

The aforementioned pointer structure is in one-to-one correspondence to an *implication graph* $IG \subset A_M \times A_M$ relating reasons to consequences, where $A_M$ is the set of bounds in $M$. $IG$ collects all references to reasons occurring in $M$ as follows: $(a, a') \in IG$ iff the occurrence of $a'$ in $M$ mentions $a$ as its reason. Given the implication graph $IG$, the set $R_{IG}(a)$ of sufficient reasons for an atom $a$ in $M$ is defined inductively as the smallest set satisfying the following three conditions.

1. $\{a\} \in R_{IG}(a)$.
2. Let $r \in R \in R_{IG}(a)$ and $S = \{q \mid (q, r) \in IG\}$.
   If $S \neq \emptyset$ then $((R \setminus \{r\}) \cup S) \in R_{IG}(a)$.
3. If $R \in R_{IG}(a)$ and $S \supset R$ then $S \in R_{IG}(a)$.

The rationale of this definition is that (1.) $a$ itself is a sufficient reason for $a$ being true, (2.) a sufficient reason of $a$ can be obtained by replacing any reason $r$ of $a$ with

a sufficient reason for $r$, (3.) any superset of a sufficient reason of $a$ is a sufficient reason of $a$.

*Conflict-Driven Learning and Non-Chronological Backtracking*

In case of a conflict encountered during the search, we can record a reason for the conflict preventing us from constructing other interval valuations provoking a similar conflict. Therefore, we traverse the implication graph $IG$ to derive a reason for the conflict encountered, and add this reason in negated form to the input formula. We use the unique implication point technique [125] to derive a conflict clause which is general in that it contains few bounds. This clause becomes asserting upon backjumping to the second largest decision level contributing to the conflict, i.e. upon undoing all decisions and constraint propagations younger than the chronologically youngest but one decision among the antecedents of the conflict.

$$\frac{\begin{array}{c}\rho'(v) = \emptyset, M = M' \cdot \langle | \rangle \cdot M'', b, b' \in M, \models \neg(b \wedge b'), \\ a_1 \in M'', a_2, \ldots, a_n \in M', \{a_1, \ldots, a_n\} \in R_{IG}(b) \cap R_{IG}(b')\end{array}}{(M, \phi) \longrightarrow (M' \cdot \langle \neg a_1 \rangle, \phi \wedge (\neg a_1 \vee \ldots \vee \neg a_n))} \tag{5.12}$$

Note that the application conditions of the rule are always satisfied when the conditions of the backtrack rule (5.9) apply, as $\rho'(v) = \emptyset$ can only arise if there are two contradicting bounds $b = v \sim c$ and $b' = v \sim' c'$ in $M$. Hence, the learning rule (5.12) can fully replace the backtrack rule (5.9). Applications of the rule are shown in figure 5.2.

Note that, while adopting the conflict detection techniques from propositional SAT solving, our conflict clauses are more general than those generated in propositional SAT solving: as the antecedents of a contraction may involve arbitrary arithmetic bounds, so do the conflict clauses. Furthermore, in contrast to nogood learning in constraint propagation, we are not confined to learning forbidden combinations of value assignments in the search space, which here would amount to learning disjunctions of interval disequations $x \notin I$ with $x$ being a problem variable and $I$ an interval. Instead, our algorithm may learn arbitrary combinations of atoms $x \sim c$, which provides stronger pruning of the search space: while a nogood $x \notin I$ would only prevent a future visit to any subinterval of $I$, a bound $x \geq c$, for example, blocks visits to any interval whose left endpoint is at least $c$, no matter how it is otherwise located relative to the current interval valuation.

### 5.3.6 Progress and Termination

The naive base algorithm described above traverses the search tree until no further splits are possible due to the search space being fully covered by conflict clauses. In contrast to purely propositional SAT solving, where the split depth is bounded by the number of variables in the SAT problem, this entails the risk of non-termination due to infinite sequences of splits being possible on each real-valued interval.

We tackle this problem by selecting a heuristics for application of the rules which guarantees a certain progress with respect to decided and deduced bounds. Therefore, we fix a progress bound $\varepsilon > 0$ (to be refined iteratively later on) and demand that the rule applications satisfy the following condition.

**Condition 5.10.** *Rules (5.7), (5.9), and (5.12) are only applied if their asserted bound $v \sim c$ narrows the remaining range of $v$ by at least $\varepsilon$, i.e. if $|c - c'| \geq \varepsilon$ for all bounds $(v \sim c') \in N$, where $N = M$ for rules (5.7) and (5.9) and $N = M'$ for rule (5.12). Rule (5.8) is only applied if both the split bound $v \sim c$ and its negation $v \not\sim c$ narrow the remaining range of $v$ by at least $\varepsilon$.*

We now define a strict partial ordering $\succ$ on the list of asserted atoms $M$.

**Definition 5.11.** Let $M = M_1|\ldots|M_n$ and $M' = M'_1|\ldots|M'_m$, where the $M_i$ and $M'_i$ represent decision levels, i.e. they consist of all bounds asserted on decision level $i$ and contain no marker symbol. Then $M \succ M'$ iff

a) $n < m$ and $\rho_{M_i} = \rho_{M'_i}$ for all $i \leq n$, or

b) $n \geq m$ and there is a $k \leq m$ such that $\rho_{M_i} = \rho_{M'_i}$ for all $i < k$ and $\rho_{M'_k}$ is a proper refinement of $\rho_{M_k}$, that is $\rho_{M'_k}(v) \subseteq \rho_{M_k}(v)$ for all variables $v$, where the inclusion is strict for at least one variable.

**Lemma 5.12.** *If $(M, \phi) \longrightarrow (M', \phi')$ by application of rule (5.6), (5.7), (5.8), (5.9), or (5.12) then $M \succ M'$.*

*Proof.* Rules (5.6) and (5.7) only modify the decision level on top of $M$ by adding a new bound which is not implied by the bounds already in $M$. Thus, after one or more applications of these rules, condition b) of definition 5.11 is satisfied, i.e. $M \succ M'$ holds. The splitting rule (5.8) adds a new decision level to $M$, that is $n$ equals $m + 1$ after application, but it does not modify already existing decision levels. Thus, by condition a) of definition 5.11, $M \succ M'$ holds. The backtracking rule (5.9) and the learning rule (5.12) both discard decision levels. Hence, after application $n \geq m$

holds. Both rules revive decision level $m$ by adding a new bound to it which is not implied by the bounds in $M_1|\ldots|M_m$. Therefore, $M \succ M'$ holds by condition b) of definition 5.11, which is satisfied for $k = m$. □

By lemma 5.12, we are able to prove termination of the algorithm if initially the intervals of all auxiliary variables and of all problem variables are bounded. For an interval $\mathbb{I}$ we define $\text{width}(\mathbb{I}) = \sup(\mathbb{I}) - \inf(\mathbb{I})$ if supremum and infimum exist, and $\text{width}(\mathbb{I}) = \infty$ otherwise.

**Lemma 5.13.** *Let $(M_0, \phi)$ be the initial state of the algorithm. Assume that all variables $v$ are bounded, i.e. $\text{width}(\rho_{M_0}(v)) \neq \infty$. Then, after finitely many applications of the rules (5.6) to (5.12) which are in compliance with condition 5.10, the algorithm reaches either 'unsat' or a state $(M, \phi')$ with $\text{width}(\rho_M(v)) \leq 2 \cdot \varepsilon$ for all $v \in BV \cup RV$, where $\varepsilon$ is the constant progress parameter of condition 5.10.*

*Proof.* Proof state 'unsat' allows no further rule applications, i.e. the algorithm stops. In the following we will therefore consider executions, where 'unsat' is *not* reached. Assume, an infinite execution $\mathcal{E} := (M_0, \phi) \rightarrow (M_1, \phi_1) \rightarrow (M_2, \phi_2) \rightarrow \cdots$ exists. Then, according to lemma 5.12, the sequence $(M_i)_{i\in\mathbb{N}}$ is strictly decreasing, i.e. $M_0 \succ M_1 \succ M_2 \succ \cdots$. Due to condition 5.10, the length of the sequence is, however, bounded by $\mathcal{O}\left(\prod_{v\in RV\cup BV} \frac{1}{\varepsilon} \cdot \text{width}(\rho_{M_0}(v))\right) < \infty$, contradicting the assumption that $\mathcal{E}$ is infinite. □

*Achieving Almost-Completeness through Restarts*

With a given progress parameter $\varepsilon$, the above procedure may terminate with inconclusive result: it may fail to terminate with an 'unsat' result, because search in conflicting branches of the search tree is stopped too early. Moreover, the interval valuation which the solver outputs after termination — we refer to it as *candidate solution box* —, possibly contains no solution. However, if $\varepsilon$ is chosen 'small enough', then any point picked from the candidate solution box will violate the constraints occuring in the formula at most by a very small amount.

To increase confidence in the result, the solver can simply be restarted with a smaller progress parameter. As all the conflict clauses are preserved from the previous run, the new run essentially only visits those interval interpretations that were previously left in an inconclusive state, and it extends the proof tree precisely at these inconclusive leaves.

By iterating this scheme for incrementally smaller progress parameter converging to zero, we obtain an 'almost complete' procedure being able to refute all *robustly* unsatisfiable (robustly satisfiable, resp.) formulae, where robustness here means that the corresponding property is stable under small perturbation of the constants in the problem. Note that such an iterative refinement of the progress parameter is considerably different from not using a progress parameter, as it still prevents infinite digression into a single branch of the search space, adding some breadth-first flavor.

## 5.4 Benchmark Results

In this section we provide experimental results obtained from benchmarking our tool HYSAT-2 which implements the ISAT algorithm, including support for all customary Boolean and arithmetic operators, in particular for sine, cosine, and exponentiation, and which is equipped with the structural optimizations for bounded model checking as explained in section 4.4.[8] A feature not yet implemented in HYSAT-2 is monotonicity relaxation for arithmetic constraints. Moreover, HYSAT-2 uses a simplified and therefore less powerful version of the deduction rule for multiplication.

The benchmarks mentioned in section 5.4.1 were performed on a 2.5 GHz AMD Opteron computer with 4 GByte physical memory, while those of sections 5.4.2 and 5.4.3 were executed on a 1.83 GHz Intel Core 2 Duo machine with 1 GByte physical memory, both running Linux.

### 5.4.1 Impact of Conflict-Driven Learning

In order to demonstrate the potential of our approach, in particular the benefit of conflict-driven learning adapted to interval constraint solving, we compare the performance of ISAT to a stripped version thereof, where learning and backjumping are disabled (but the optimized data structures, in particular watched atoms, remain functional).

We consider bounded model checking problems, that is, proving a property of a hybrid discrete-continuous transition system for a fixed unwinding depth $k$. Without learning, the interval constraint solving system failed on every moderately interesting hybrid system due to complexity problems exhausting memory and runtime. This

[8] A HYSAT-2 executable, a manual, and the input files for the benchmarks can be found on http://hysat.informatik.uni-oldenburg.de.

is not surprising, because the number of boxes to be potentially visited grows exponentially in the number of variables occuring in the constraint formula, which in turn grows linearly in both the number of problem variables in the hybrid system and in the unwinding depth $k$. When checking a model of an *elastic approach to train distance control* (cf. p. 73), the version without learning exhausts the runtime limit of 3 days already on unwinding depth $1$, where formula size is $140$ variables and $30$ constraints. In contrast, the version with conflict-driven learning solves all instances up to depth $10$ in less than 3 minutes, thereby handling instances with more than $1100$ variables, a corresponding number of triplets and pairs, and $250$ inequality constraints. For simpler hybrid systems, like the model of a *bouncing ball* falling in a gravity field and subject to non-ideal bouncing on the surface, the learning-free solver works due to the deterministic nature of the system. Nevertheless, it fails for unwinding depths $> 11$, essentially due to enormous numbers of conflicting assignments being constructed (e.g., $> 348 \cdot 10^6$ conflicts for $k = 10$), whereas learning prevents visits to most of these assignments (only 68 conflicts remain for $k = 10$ when conflict-driven learning is pursued). Consequently, the learning-enhanced solver traverses these problems in fractions of a second; it is only from depth $40$ that our solver needs more than one minute to solve the bouncing ball problem ($2400$ variables, $500$ constraints). Similar effects were observed on chaotic real-valued maps, like the *gingerbread map*. Without conflict-driven learning, the solver ran into approx. $43 \cdot 10^6$, $291 \cdot 10^6$, and $482 \cdot 10^6$ conflicts for $k = 9$ to $11$, whereas only $253$, $178$, and $155$ conflicts were encountered in the conflict-driven approach, respectively. This clearly demonstrates that conflict-driven learning is effective within interval constraint solving: it dramatically prunes the search space, as witnessed by the drastic reduction in conflict situations encountered and by the frequency of backjumps of non-trivial depth, where depths of $47$ and $55$ decision levels were observed on the gingerbread and bouncing ball model, respectively. Similar effects were observed on two further groups of benchmark examples: an oscillatory *logistic map* and some geometric decision problems dealing with the intersection of $n$-dimensional geometric objects. On random formulae, we even obtained backjump distances of more than $70000$ levels. The results of the aforementioned benchmarks, excluding the random formulae, are presented in figure 5.5.

### 5.4.2 Comparison to ABSOLVER

Next, we provide a comparison to ABSOLVER [14], which, to the best of our knowledge, is the only other SMT-based solver addressing the domain of large Boolean

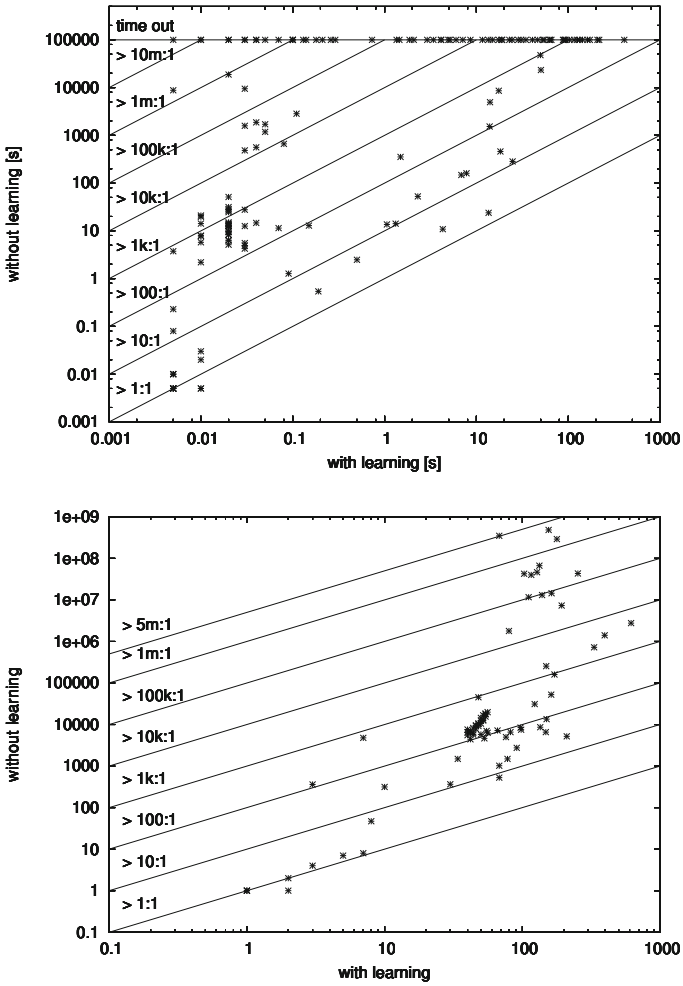


Figure 5.5: Performance impact of conflict-driven learning and non-chronological backtracking: runtime in seconds (top) and number of conflicts encountered (bottom).

combinations of nonlinear arithmetic constraints over the reals. The currently reported implementation [14] uses the numerical optimization tool IPOPT [119] for solving the nonlinear constraints. IPOPT is a highly efficient tool for numerical local optimization. However, in contrast to global optimization methods, it only searches for local solutions, and hence may incorrectly claim a satisfiable set of constraints inconsistent. Moreover, IPOPT may also produce incorrect results due to rounding errors. Note that solving nonlinear constraints globally and without rounding errors is considered a problem that is harder to solve by orders of magnitude.

The current implementation of ABSOLVER[9] supports the arithmetic operations of addition (and subtraction) and multiplication (and division) only. Therefore, the respective benchmarks are restricted to polynomial arithmetic. We performed the experiments on ABSOLVER with options `sat:zchafflib`, `l:coin`, `nl:ipopt`.

Table 5.6 on the following page lists the experimental results. The second column contains the unwinding depth of bounded model checking problems, the third column the number of arithmetic operators in the benchmark.

The first benchmark is described in [79, p. 5]. Benchmarks 2 to 4 can be found in [14] as well as on ABSOLVER's web page. The industrial case study of a mixed-signal circuit in a car-steering control system (benchmark 5) is described in [14], yet is not publicly available due to protection of intellectual property. The remaining benchmarks are bounded model checking problems of hybrid systems and of iterated chaotic maps. Except for the car-steering control system, all benchmarks are available from the HYSAT-2 website.

For the first 4 benchmarks, which are very small numeric CSPs without complex Boolean structure, the runtimes are almost equal. For all other benchmarks, iSAT yields orders of magnitude of speedup compared to ABSOLVER, no matter whether the benchmarks feature moderately complex Boolean structure, like the mixed-signal circuit in car-steering, feature extremely complex Boolean structure, as in bounded model-checking of the linear hybrid automata h3_train, renault_clio, and aircraft and the nonlinear bouncing ball, or are almost conjunctive (basically, one disjunction per unwinding step), like the iterated duffing and tinkerbell maps. The comparable performance on purely conjunctive problems, which is indicative of the relative performance of the underlying arithmetic reasoning engines, together with the huge performance gap on problems with more complex Boolean structure shows that the tight integration of Boolean and arithmetic constraint propagation pursued in iSAT saves overhead incurred in an SMT approach deferring theory problems to subordinate theory solvers.

[9] Available from http://absolver.sourceforge.net.

| Benchmark | BMC depth | ♯arith_op | iSAT | ABSOLVER | speedup |
|---|---|---|---|---|---|
| nonlinear_CSP | — | 44 | 0m0.032s | 0m0.072s | 2.2 |
| esat_n11_m8_nonlinear | — | 7 | 0m0.028s | 0m0.012s | 0.4 |
| nonlinear_unsat | — | 2 | 0m0.004s | 0m0.032s | 8.0 |
| div_operator | — | 1 | 0m0.004s | 0m0.028s | 7.0 |
| car_steering | — | 138 | 0m0.268s | 2m11.032s | 488.9 |
| h3_train | 2 | 102 | 0m0.244s | 0m2.968s | 12.2 |
| h3_train | 3 | 153 | 0m0.304s | 0m6.480s | 21.3 |
| h3_train | 4 | 204 | 0m0.344s | 0m10.401s | 30.2 |
| h3_train | 5 | 255 | 0m0.348s | 0m15.981s | 45.9 |
| h3_train | 17 | 867 | 0m30.718s | 3m22.769s | 6.6 |
| h3_train | 18 | 918 | 0m33.346s | 3m39.374s | 6.6 |
| h3_train | 30 | 1530 | 0m46.519s | 10m55.965s | 14.1 |
| renault_clio | 2 | 132 | 0m0.020s | 0m0.764s | 38.2 |
| renault_clio | 3 | 198 | 0m0.024s | 0m1.628s | 67.8 |
| renault_clio | 30 | 1980 | 0m0.300s | 3m48.158s | 760.5 |
| renault_clio | 31 | 2046 | 0m0.344s | 4m9.528s | 725.4 |
| aircraft | 5 | 132 | 0m0.044s | 2m30.113s | 3,411.7 |
| aircraft | 6 | 157 | 0m0.056s | 5m47.182s | 6,199.7 |
| aircraft | 10 | 257 | 0m1.496s | 50m43.594s | 2,034.5 |
| duffing_map | 3 | 21 | 0m0.004s | 0m4.904s | 1,226.0 |
| duffing_map | 4 | 28 | 0m0.004s | 1m58.571s | 29,642.7 |
| duffing_map | 5 | 35 | 0m0.004s | 3m35.117s | 53,779.2 |
| duffing_map | 6 | 42 | 0m0.004s | 5m52.822s | 88,205.5 |
| duffing_map | 7 | 49 | 0m0.001s | 0m22.313s | 22,313.0 |
| duffing_map | 8 | 56 | 0m0.004s | 1m16.849s | 19,212.2 |
| duffing_map | 20 | 140 | 0m0.008s | 6m15.675s | 46,959.4 |
| duffing_map | 30 | 210 | 0m0.012s | $>$ 60m | $>$ 300,000 |
| tinkerbell_map | 1 | 22 | 0m0.004s | 0m1.292s | 323.0 |
| tinkerbell_map | 2 | 38 | 0m0.008s | 3m12.052s | 24,006.5 |
| tinkerbell_map | 3 | 54 | 0m0.048s | 7m58.210s | 9,962.7 |
| tinkerbell_map | 4 | 70 | 0m0.676s | 15m49.495s | 1,404.6 |
| tinkerbell_map | 5 | 86 | 0m1.080s | 27m37.548s | 1,534.8 |
| tinkerbell_map | 6 | 102 | 0m0.644s | 46m42.739s | 4,352.1 |
| nonlinear_ball | 2 | 52 | 0m0.008s | 9m8.538s | 68,567.2 |
| nonlinear_ball | 3 | 78 | 0m0.008s | $>$ 60m | $>$ 450,000 |
| nonlinear_ball | 4 | 104 | 0m0.012s | $>$ 60m | $>$ 300,000 |

Table 5.6: Performance of iSAT relative to ABSOLVER.

### 5.4.3 Comparison to HYSAT-1

We also did a comparison of HYSAT-2 with our LP-based solver HYSAT-1 (cf. chapter 4), using BMC benchmarks comprising linear arithmetic only. HYSAT-2 passed the back-to-back test with HYSAT-1 successfully: Given enough time and memory, HYSAT-2 was able to solve all benchmarks with the expected result. In particular, it found error traces at the same unwinding depth as HYSAT-1, but not earlier, as could have been expected, since HYSAT-2 may fail to detect unsatisfiability of a formula and instead report a candidate solution box (i.e. an 'almost-solution'). However, we also observed, that some problems which were relatively easy to solve for the linear programming based solver, turned out to be hard for HYSAT-2. An example is the train distance control benchmark described on page 73. We quote some statistics obtained from a version involving three trains, which has an error trace of 22 steps. HYSAT-1 solves all 23 BMC instances in a total time of 5.3 seconds, thereby invoking the linear programming solver 2235 times. As opposed to this, HYSAT-2 needs 139 minutes to complete all 23 instances, thereby performing 173500 decisions (i.e. splitting steps), analyzing 100628 conflicts, and deducing more than $2.2 \cdot 10^9$ bounds. Most of the time is spent on instances near the satisfiability threshold, i.e. for BMC depths near the point where the unwound formula undergoes the phase transition from being unsatisfiable to satisfiability. While HYSAT-2 solves unwinding depths $0 \leq k \leq 19$ in a total of 35.3 seconds, it needs 11.4 minutes for $k = 20$, 122.2 minutes to prove unsatisfiability for $k = 21$, and 4.8 minutes for finding a candidate solution box for $k = 22$. The slowdown compared to HYSAT-1 was not unexpected, since it is well-known that linear programming provides a much stronger deduction mechanism than interval constraint propagation. In fact, deduction based on ICP is fairly incomplete (and has therefore to be complemented by interval-splitting to make a usable solving algorithm), which can be demonstrated by the following example. Consider the formula $x + y = 0 \wedge x - y = 0$, where $x, y \in [-1, 1]$. Obviously, $x = y = 0$ is the only solution of the formula, yet, the deduction rules given in section 5.3.3 fail do tighten the intervals of $x$ and $y$. Given this, it seems advisable to combine the ISAT algorithm with linear programming in order to accelerate solving of problems involving linear arithmetic.

## 5.5 Reachability Analysis with HYSAT-2: a Case Study

To demonstrate the use of HYSAT-2, we present a concrete application benchmark from the transportation domain. The model was generated using the MATLAB/SIMU-

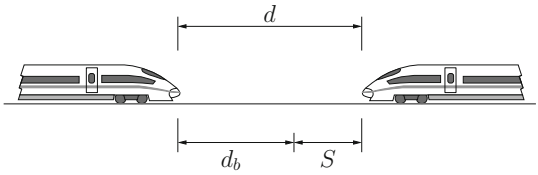


Figure 5.6: The absolute braking distance $d$ equals the sum of the braking distance $d_b$ of the following train and an additional safety distance $S$.

LINK tool. A structure-driven and compositional, currently manually applied translation — of which we show the most interesting aspects in the second subsection — then allows for fully automatic bounded model checking. The retrieved error trace is subsequently shown in a side-by-side comparison with a simulation run of the SIMULINK model.

### 5.5.1 ETCS Model

The benchmark deals with analyzing the safety of a railway system when operated under a *moving block* principle of operation. In contrast to conventional interlocking schemes in the railway domain, where static track segments are locked in full, the moving block principle applies headway control as required by the braking distance, reserving a moving block ahead of the train depending on speed and braking capabilities. There are two variants of this principle, namely train separation in relative braking distance, where the spacing of two following trains depends on the current speeds and braking capabilities of both trains, and train separation in absolute braking distance, where the distance of two following trains equals the braking distance of the second train plus an additional safety distance (figure 5.6). Within this case study we apply the second variant which will also be used in the forthcoming European Train Control System (ETCS) Level- 3. We consider an abstract model of ETCS Level 3. Within this simplified version, all trains operate in obedience of the following procedures and regulations:

- All trains run on the track travel in the same direction. In particular, the train sequence is fixed (no overtaking) and a train must not change its direction.
- Each train broadcasts the position of its end to the following train every 8 seconds via radio.

- Whenever a train receives an update of the position of the train running ahead, it computes its *movement authority* $m$, i.e. the stopping point it must not cross, and the deceleration $a$ which is required to meet that stopping point. These are computed according to the formulae

$$m = xr - (xh + S) \quad \text{and} \quad a = \frac{v^2}{2m}$$

  where $xr$ is the position of the rear end of the first train, $xh$ is the position of the head of the second train, and $v$ is its velocity.

  Braking is automatically applied whenever the value of $a$ exceeds a certain threshold $b_{\text{on}}$. Automatic braking ends if $a$ falls below $b_{\text{off}}$.

- When a train is not in automatic braking mode, acceleration and deceleration are freely controlled by the train operator within the physical bounds of the train.

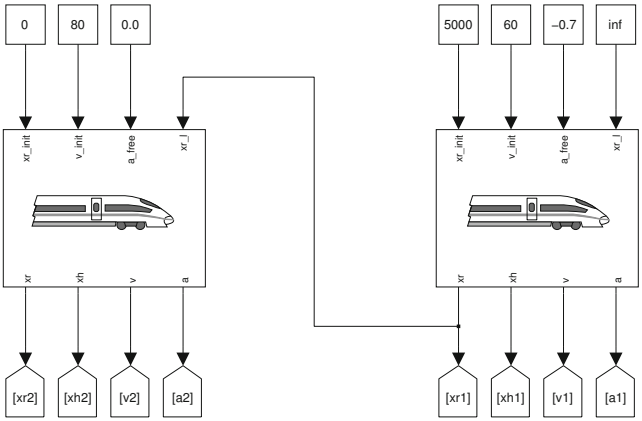


Figure 5.7: Top-level view of the MATLAB/SIMULINK model.

Figure 5.7 shows the top-level view of the MATLAB/SIMULINK implementation of the model in a version with two trains. Inputs of a train block are the initial position of the train, its initial speed, the acceleration applied in free-running mode and the position of the rear end of the train which is running ahead. Outputs are the position of the rear end of the head of the train, its velocity and current acceleration.

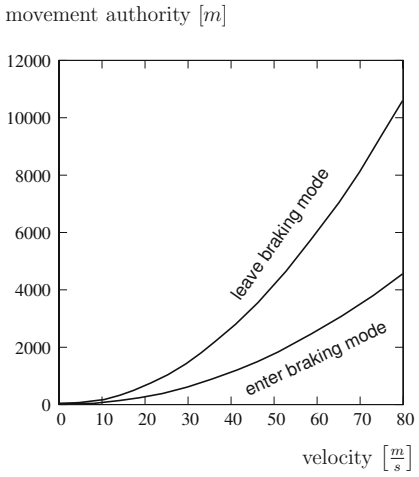


Figure 5.8: Switching curves of the controller.

| Parameter | | Value |
|---|---|---|
| length of the train | [m] | 200 |
| maximum velocity | [m/s] | 83.4 |
| maximum acceleration | [m/s$^2$] | 0.7 |
| maximum deceleration | [m/s$^2$] | -1.4 |
| $b_{\text{on}}$ | [m/s$^2$] | -0.7 |
| $b_{\text{off}}$ | [m/s$^2$] | -0.3 |
| safety distance $S$ | [m] | 400 |

Table 5.7: Parameters of the ETCS case study.

The implementation of a train block is shown in figure 5.9 on the next page. The parameters of the model are given in table 5.7. We chose them to roughly match the characteristics of an ICE 3 half-train. The switching curves of the controller resulting from the choice of $b_{\text{on}}$ and $b_{\text{off}}$ are shown in figure 5.8. When crossing the lower curve from above, automatic braking is applied because from this point on a deceleration of at least $b_{\text{on}}$ is required to stop the train within the movement authority. Automatic braking is released when the upper switching curve is traversed from below.

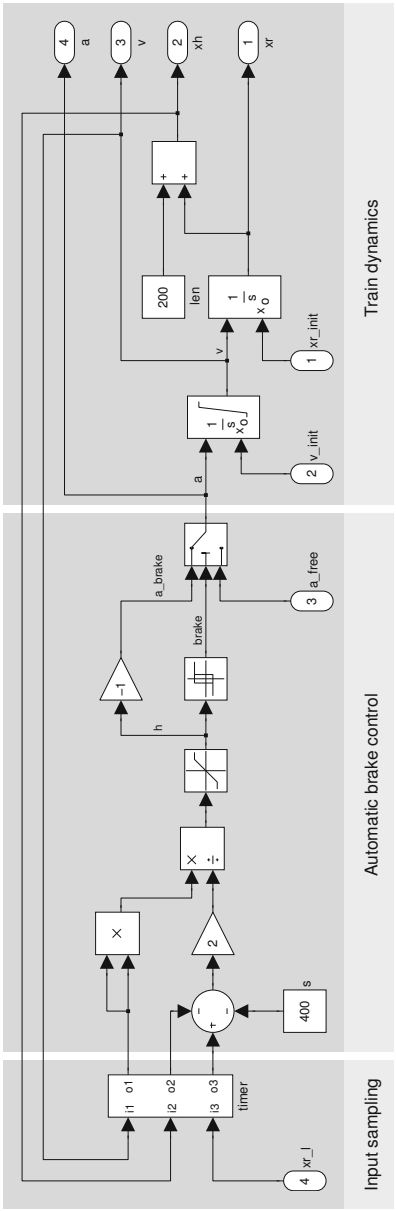


Figure 5.9: Implementation of the controller and train dynamics.

The sample trace in figure 5.10 a) on page 129, which was obtained by simulation of the model, seems to suggest that the controller works correctly: The trains start with an initial distance of 5000 m, the second train being 20 m/s faster than the first train, which is braking with a deceleration of -0.7 m/s$^2$. The second train automatically starts braking, adjusting its deceleration in intervals of 8 seconds, and comes to stop exactly 400 m behind the first train.

Instead of performing a potentially unlimited number of simulations to cover all possible traces of the system, we encode the model for HYSAT-2. This allows us to check all traces (up to a certain unwinding depth) for collisions of the trains without having to guess scenarios for the open inputs that may lead to these unsafe states.

```
1  DECL
2      define f = 2.0;
3      float [0, 1000] x;
4      boole jump;
5
6  INIT
7      x = 0.6;
8      !jump;
9
10 TRANS
11     jump' <-> !jump;
12     jump -> f * x' = x;
13     !jump -> x' = x + 2;
14
15 TARGET
16     x > 3.5;
```

Listing 5.1: Sample input file for HYSAT-2.

### 5.5.2 Encoding into HySAT

In order to encode the model described above, we first introduce the input language of the HYSAT-2 tool by means of the sample input file shown in listing 5.1. The file consists of four parts[10]:

- `DECL`: All variables used in the mode have to be declared here. Types supported by HYSAT-2 are `float`, `int`, and `boole`. For float and integer variables a bounded range has to be specified. It is also possible to define symbolic constants in this section (see the definition of `f` in line 2).

[10]Note, that HYSAT-2 has a second file format for solving single formulae. See the manual for details.

- `INIT`: This part contains a formula describing the initial state(s) of the system to be investigated. In the example file, `x` is initialized to 0.6, and `jump` is set to `false`, since this is the only valuation which satisfies the constraint `!jump`, where '`!`' stands for '`not`'.

- `TRANS`: This formula describes the transition relation of the system. Variables may occur in primed or unprimed form. A primed variable represents the value of that variable in the successor step, i.e. after the transition has taken place. Thus, line 13 of the example states, that if `jump` is `false` in the current state, then the value of `x` in the next state is given by its current value plus 2. The semicolon which terminates each constraint can be read as an AND-operator. Hence, `TRANS` is a conjunction of three constraints.

- `TARGET`: This formula characterizes the state(s) whose reachability is to be checked. In the example, we want to find out if a state is reachable in which `x > 3.5` holds.

Note, that the file sections `INIT`, `TRANS`, and `TARGET` just correspond to the components $I(\vec{y}^{\,0})$, $T(\vec{y}\,,\vec{y}\,')$, and $R(\vec{y}^{\,k})$ of the BMC formula constructed by the encoding schemes given in chapter 2 (cf. page 26ff).

When calling HYSAT-2 with the input file above, it successively unwinds the transition relation $k = 0, 1, 2, \ldots$ times, conjoins the resulting formula with the formulae describing the initial state and the target states, and thereafter solves the formula thus obtained. For $k = 0, 1, 2, 3, 4$, the formulae are all unsatisfiable, for $k = 5$ however, a solution is found. The output generated by HYSAT-2 for $k = 4$ and $k = 5$ is given in listing 5.2 on the following page. For $k = 5$, HYSAT-2 reports the values of `jump` and `x` for each step of the unwound transition relation. After the last transition, as required, `x > 3.5` holds.

If HYSAT-2 terminates with the result '`unsatisfiable`', then the formula is actually unsatisfiable. If the solver stops with the result '`candidate solution box found`', then the solver could not detect any conflicts within the reported intervals. Recall, however, that this does *not* mean that the intervals are guaranteed to actually contain a solution. Nevertheless, the sizes of the returned intervals do not exceed a user-specified parameter $\varepsilon$. From a practical point of view, this means that the solver returns a solution with precision $\varepsilon$.

```
SOLVING:
    k = 4

RESULT:
    unsatisfiable

SOLVING:
    k = 5

RESULT:
    candidate solution box found

SOLUTION:
    jump (boole):
        @0: [0, 0]
        @1: [1, 1]
        @2: [0, 0]
        @3: [1, 1]
        @4: [0, 0]
        @5: [1, 1]

    x (float):
        @0: [0.6, 0.6]
        @1: [2.6, 2.6]
        @2: [1.3, 1.3]
        @3: [3.3, 3.3]
        @4: [1.65, 1.65]
        @5: [3.65, 3.65]
```

Listing 5.2: Snippet of solver output.

For encoding the MATLAB/SIMULINK model of the ETCS case study (cf. figure 5.9 and 5.7) we first introduce a variable (of a corresponding type and domain) for each connecting line of the SIMULINK model and declare them in the `DECL` part. (Please note that by substitution of the invariants of some connected MATLAB/SIMULINK blocks, we may save introduction of variables for some lines.) In the `INIT` part we require that the trains are stopped and their distance is $1000$ meters. Contrary to the initial state of the simulation, the initial values of the accelerations are not fixed but may be chosen freely from their domain $[-1.4, 0.7]$. For the translation of the SIMULINK blocks into the `TRANS` part of HYSAT-2 we illustrate the encodings of the most interesting blocks of figure 5.9, i.e. the relay, switch, and integrator blocks. Simpler blocks, e.g. the sum block, can be encoded straightforwardly, e.g. by `o = i1 + i2` where `o` is the output and `i1, i2` are the inputs of the sum block. The predicative encodings of all blocks are conjoined by logical conjunction, represented by a semicolon in concrete HYSAT-2 syntax.

- *Relay block.* When the relay is 'on' (indicated by the Boolean variable `is_on`), it remains 'on' until the input drops below the value of the switch-off-point parameter `param_off`. When the relay is 'off' (i.e. not `is_on` or `!is_on` holds), it remains 'off' until the input exceeds the value of the switch-on-point parameter `param_on`. The switch-on/off-point parameters are defined as symbolic constants in the `DECL` part, i.e. `define param_on = 0.7;` and `define param_off = 0.3;`.

```
( is_on and h >  param_off) -> ( is_on' and  brake);
( is_on and h <= param_off) -> (!is_on' and !brake);
(!is_on and h <  param_on ) -> (!is_on' and !brake);
(!is_on and h >= param_on ) -> ( is_on' and  brake);
```

- The *switch block* passes through the first input `a_brake` or the third input `a_free` based on the value of the second input `brake`.

```
 brake -> a = a_brake;
!brake -> a = a_free;
```

- *Integrator block with saturation.* The potentially new value `v'` of the velocity is determined by an Euler approximation with sampling time `dt` $= 8, 2$, and $1$ seconds for the encodings A, B, and C, respectively, and stored temporarily in the auxiliary variable `aux`. According to the saturation parameters, `v'` is set to its value as shown below. The lower and upper saturation limits are $0.0$ and `v_max` $= 83.4$, respectively.

```
aux = v + dt * a;
aux <= 0.0                    -> v' = 0.0;
aux >= v_max                  -> v' = v_max;
(aux > 0.0 and aux < v_max) -> v' = aux;
```

  Note that other (exact or safe) approximation methods are applicable here. For the sake of clarity, we opt for the simple, in general inexact, Euler method. We refer the reader to section 2.2.2 for approaches to *safely* approximate nonlinear continuous behaviour.

Finally, completing the HySAT-2 input we specify a target state, i.e. an undesired property of the system to be checked. In our case study, we want to know whether the controller is incorrect in the sense that collisions of the trains are possible. Hence,

we add the formula `xr1 - (xr2 + length) <= 0.0;` to the `TARGET` section, meaning that the distance of the rear position of the first train `xr1` and the head position of the second train, i.e. rear position `xr2` plus length of the train, is less than or equal zero.

An automatic translation of a subset of SIMULINK models to HYSAT-2 has been implemented in [97]. This translation follows the scheme sketched above. While not currently being able to translate the above model due to some of its SIMULINK blocks not being supported, it is supposed to cover all these blocks as well as a representative subset of STATEFLOW statecharts, as embedded into SIMULINK, in the future.

### 5.5.3 Results

Running HYSAT-2 on the encoded models yields error traces of lengths 8 for encoding A, 33 for encoding B, and 66 for encoding C. Bounded model checking thus revealed a simple bug of the controller that was yet subtle enough not to be noticed when designing the model: If the moving authority $m$ becomes zero or even negative (which may happen since the controller re-computes the deceleration setting only every 8 seconds), then instead of applying the maximum braking force, the controller switches back to free-running mode, allowing the operator of the train to accelerate and crash into the rear of the train ahead. While the simulation run depicted in figure 5.10 a) suggests that the distance controller works as intended, the error trace shown in b), which was obtained from solving encoding C, exposes the bug.

The experiments were performed on a 2.5 GHz Opteron machine with 4 GByte physical memory, running Linux. The total runtimes for solving all BMC instances up to the error trace were about 10 seconds for encoding A (with sampling time `dt` $= 8$ seconds), 1.8 minutes for encoding B (`dt` $= 2$ seconds) and 21.5 minutes for encoding C (`dt` $= 1$ second). The runtime largely depends on the solver settings, e.g. the splitting heuristics chosen, with the runtimes reported above being the best we could obtain for the respective encoding.

The diagrams in figure 5.11 show the impact of conflict-clause sharing and BMC-specific decision strategies, as discussed in section 4.4, on the runtime of the solver. The measurements were taken using encoding C. The graphs plot the accumulated runtime for solving the first $n$ BMC instances against $n$, where $0 \leq n \leq 65$. For example, HYSAT-2 solves the first 40 unwindings of encoding C in about 10 seconds if sharing of conflict clauses is enabled. Without sharing, HYSAT-2 needs more than 100 seconds to complete them all. Concerning decision strategies, the forward strat-

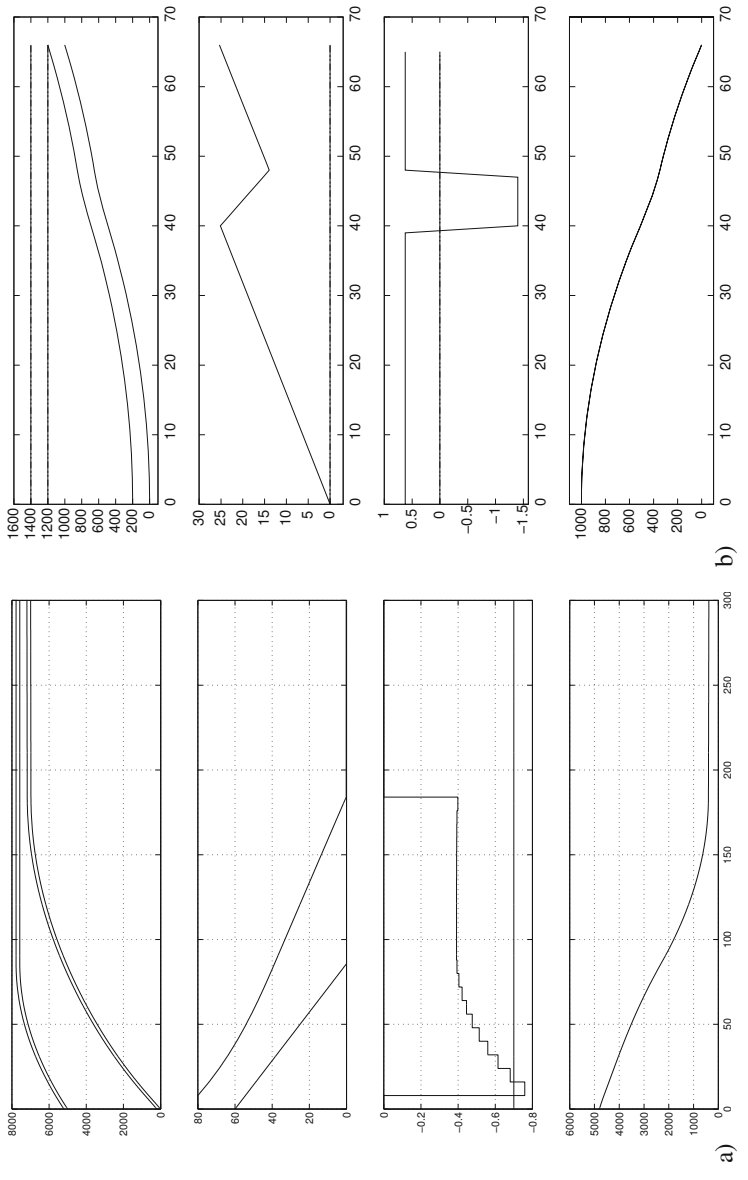


Figure 5.10: a) Simulation run of the SIMULINK model with fixed parameters. From top to bottom the charts show the positions, speeds, accelerations and distance of the two trains over the simulated time. b) Error trace found by HYSAT-2.

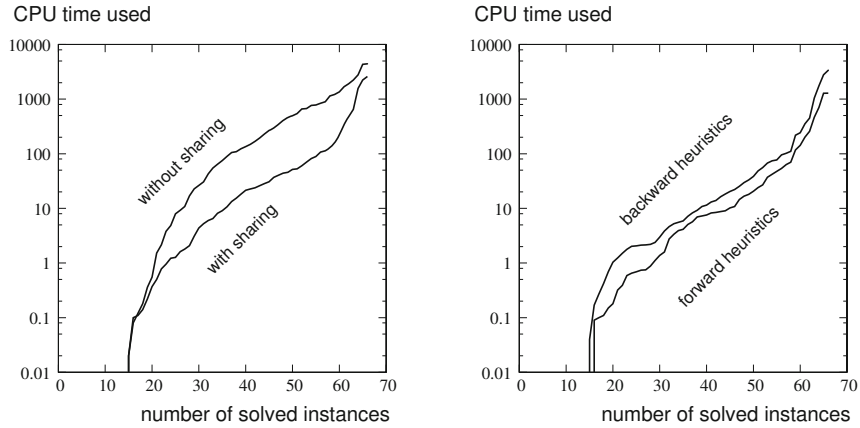


Figure 5.11: Impact of BMC-specific optimizations.

egy performs significantly better than the backward strategy. Isomorphy inference did not yield performance gains for this specific benchmarks.

The BMC formula solved to find the trace shown in figure 5.10 contains 7809 variables, thereof 4901 of type float, 2709 of type Boolean, and 2908 of type integer. (The latter are used to model the timer subsystem appearing in figure 5.9.) Solving all 66 BMC instances required 69969 decisions, 27047 conflicts, and roughly $5.8 \cdot 10^8$ deductions of bounds in total.

## 5.6 Discussion

Within this chapter, we have demonstrated how a tight integration of DPLL-style SAT solving and interval constraint propagation can reconcile the strengths of the SAT-modulo-theory (SMT) approach with those of interval constraint propagation (ICP). The particular strength of SMT in manipulating large and complex-structured Boolean combinations of constraints over a (in general decidable) theory is thus lifted to the undecidable domain of nonlinear arithmetic involving transcendental functions. In particular, we were thus able to canonically lift to interval-based arithmetic constraint solving of massively disjunctive constraint problems the crucial algorithmic enhancements of modern propositional SAT solvers, especially lazy clause evaluation, conflict-driven learning, and non-chronological backtracking. Our benchmarks

demonstrate significant performance gains up to multiple orders of magnitude compared to a pure backtrack SMT+ICP algorithm. Equally important, the performance gains were consistent throughout our set of benchmarks, with only one trivial instance incurring a negligible performance penalty due to the more complex algorithms. Similar results were observed in comparison with a nonlinear SMT solver (ABSOLVER) employing classical deferring of theory problems to subordinate solvers, which substantiates the argument that tighter integration of DPLL and ICP is beneficial.

# 6 Conclusion

*A great while ago the world began · With hey, ho, the wind and the rain · But that's all one, our play is done · And we'll strive to please you every day.*

*William Shakespeare*

## 6.1 Achievements

Motivated by the need of solver technology for formal verification of hybrid discrete-continuous systems, we have investigated algorithms for solving formulae which are quantifier-free Boolean combinations of arithmetic constraints over the reals. Our contributions to the state of the art in the field are, in brief, as follows:

- We have demonstrated that acceleration techniques employed in modern propositional SAT solvers, in particular lazy clause evaluation, learning, and backjumping, generalize smoothly to DPLL-like procedures for solving conjunctions of pseudo-Boolean constraints, a much more succinct language for expressing Boolean functions than CNF.

- We have investigated how to efficiently couple a DPLL-based SAT solver with a linear programming routine in a DPLL($T$) framework in order to obtain a solver which is tailored for BMC of hybrid systems with linear continuous dynamics. To this end, we took advantage of BMC-specific optimizations previously only employed in pure propositional solvers, and demonstrated that such optimizations, in particular sharing and isomorphic copying of conflict clauses, are even more effective in solvers with support for real-valued arithmetic, since the computational costs for conflict analysis are much higher in this domain.

- We have conceived a tight integration of the DPLL procedure for Boolean SAT solving with interval constraint solving. The resulting algorithm, called iSAT, generalizes the DPLL routine and is capable of solving Boolean combinations of nonlinear arithmetic constraints which may even involve transcendental functions. We have demonstrated that our approach can deal with formulae involving some ten thousands of Boolean and real-valued variables.

C. Herde, *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure*, DOI 10.1007/978-3-8348-9949-1_6,

All algorithms proposed in the preceding chapters have been implemented and evaluated using benchmarks from the envisaged application domain. Our work has been presented at international conferences and has been published in renowned scientific journals. Moreover, our contributions have been successfully evaluated by independent experts from the field during the review meeting of the Transregional Collaborative Research Center 'Automatic Verification and Analysis of Complex Systems' (AVACS), funded by the DFG, in September 2007.

The relevance of our work for industrial applications is witnessed by inquiries from *Airbus UK*[1], who requested the source code of HYSAT-2, and *Toyota Research & Development*[2], who evaluated the tool for the purpose of test pattern generation for an anti-lock braking system. *Airbus UK* has meanwhile allocated funding for an integration of ISAT into the KODKOD constraint solver for relational logic [116] in order to equip the latter with numerical reasoning capabilities. Furthermore, *Airbus UK*[3] is currently evaluating the verification of MATLAB/SIMULINK designs with HYSAT-2.

Four ongoing PhD projects which investigate different extensions of the ISAT algorithm give further evidence to the potential of the ISAT approach. See [53] for a summary of these endeavours, which are carried out at the universities of Freiburg and Oldenburg within the AVACS project.

While HYSAT-1 has predominantly been used internally in AVACS, HYSAT-2 is publicly available since 2007 and has meanwhile been used by a number of researchers for applications in various domains, e.g. for the analysis of signaling pathways in biomolecular systems [105], for determination of bit-widths for finite precision implementation of numerical calculations [81], and for the analysis of DC operating points in analogue circuits [122]. Within the research project DEMS, conducted by the *OFFIS Institute for Information Technology*, HYSAT-2 was employed to analyze the risk of blackouts due to cascading failures in electric power distribution networks.

Moreover, HYSAT-1 and HYSAT-2 have been used within numerous student projects at the *University of Oldenburg* and at the *Technical University of Denmark*, e.g. for inductive verification of hybrid systems [52, 106], for solving task allocation and scheduling problems, for the computation of worst-case execution times, for verification of a parking-assistance system, and for motion planning of a parallel robot.

[1] Marcelin Fortez Da Cruz, Airbus UK Systems Engineering, Bristol, UK. Personal communication (email), March 31, 2008.

[2] Masakazu Adachi, Toyota Central R&D Labs., Inc., Nagakute, Aichi, Japan. Personal communication (meeting in Oldenburg), February 16, 2009.

[3] Internship project 'Hybrid function analysis using SAT/interval arithmetic constraint solving' carried out at *Airbus UK* in conjunction with the *University of Edinburgh*, started in September 2009.

The variety of different applications clearly demonstrates the versatility of tools like those we dealt with in this thesis.

## 6.2 Perspectives

It seems to be an invariant truth of solver development that with every solver finished, many ideas on how to do even better pop up. What just appeared to be the ultimate engine, turns out to be the precursor of a presumably much more powerful algorithm only. Hoping that solver evolution keeps up its current speed, we provide some starting points for future work.

*Improving Performance and Usability of* HYSAT-2

The primary objective of our HYSAT-2 implementation was to evaluate the concepts described in chapter 5, rather than to provide the most efficient code implementing them. HYSAT-2 is an academic prototype which still lacks several important features, which would probably make the tool much faster and improve its usability. We list some extensions which we consider important in this regard, although they rather call for an implementation effort than for research.

- *Improve performance on Boolean problems.* The ISAT algorithm employs the same algorithmics for solving Boolean problems as a propositional SAT solver does. Yet, HYSAT-2 would drastically fail when compared against such a tool on purely Boolean benchmarks. The main reason is that the data structures used in HYSAT-2 are fairly general and not optimized w.r.t. Boolean variables. Boolean, integer, and real-valued variables are internally represented by the same type of object which makes handling of Boolean subproblems much less efficient than it could be. A future reimplementation should improve on this.

- *Accuracy measure for almost-solutions.* In case that HYSAT-2 cannot prove an input formula to be unsatisfiable, it provides a small box in the search space which contains if not genuine solutions, then at least almost-solutions, which violate arithmetic constraints occuring in the formula at most by a very small amount. Besides providing the interval valuation defining this box, HYSAT-2 should select an arbitrary point from each interval, report it to the user, preferrably in form of an exact rational value, and compute the violation of all equalities and inequalities under this specific valuation. This would allow to

assess whether the accuracy achieved is sufficient or whether the solver should be restarted with a smaller progress parameter.

- *Individual progress parameter for each variable.* The progress parameter $\varepsilon$ used to enforce termination of the ISAT algorithm in particular determines the width of the intervals defining the candidate solution box delivered by the solver. It therefore allows to control the accuracy of almost-solutions computed by ISAT. In the current implementation the same $\varepsilon$ applies to all variables, even though the required accuracy might vary for different variables. Given that engineers are usually well aware of the acceptable tolerances for the various quantities occuring in their models, we believe that it would improve the usability of HYSAT-2, if the progress parameter (and thus the precision of the result) could be specified individually for each variable.

- *Integration of linear programming.* The comparison of HYSAT-2 and HYSAT-1 in section 5.4.3 shows that ISAT's ICP-based reasoning can, in general, not compete with linear programming. To accelerate reasoning for linear constraints, it is thus recommendable to integrate linear programming into ISAT. Since ISAT is a generalization of the DPLL procedure, the integration can in principle be carried out in DPLL($T$)-style, as described in chapter 4, opening up a variety of design options, e.g. whether to handle linear constraints only by linear programming or to additionally decompose them into definitions and apply ICP, and when to call the LP solver: before, during or after ICP. In the following section about linear relaxations of nonlinear constraints, we describe another form of integration which we consider superior to a DPLL($T$) approach.

- *Spezialized deduction rules for, e.g., pseudo-Boolean constraints.* For very frequent types of constraints it might be worthwhile to provide specialized contractors which directly handle non-decomposed constraints and thus allow to save the auxiliary variables needed for decomposition otherwise. For pseudo-Boolean constraints, e.g., one could use the propagation rule given in section 3.3.1, and in addition take advantage of the watched literal scheme explained in section 3.3.2, instead of decomposing and processing such constraints, using the standard deduction rules of ISAT.

- *Splitting heuristics.* The current version of HYSAT-2 selects variables for splitting in a simple round robin fashion, using a static variable order. So far, hardly any effort has been made to adapt, implement, and evaluate more elaborated,

dynamic schemes for decision making, like those successfully employed in state-of-the-art propositional SAT solvers. Here, a wide field for experiments opens. One could e.g. try heuristics which prefer variables for splitting which actively contributed to recent conflicts, variables whose splitting has triggered many deductions in the past, or variables whose interval is still large compared to others. Currently, HYSAT-2 performs midpoint-splitting of intervals only. An extension, which seems worthwhile to be explored, is to derive suitable splitting points (or bounds) from the formula to be solved and store them, for each variable, in a list which is consulted when the respective variable is chosen for splitting. Assume, for example, that the input formula contains the constraint $\cos(x) + y^2 \geq 4$. Then one could add '$\geq 4$' to the list associated with the auxiliary variable representing the left-hand side of the constraint. By using this bound or its negation for splitting, the solver could, just like in a DPLL($T$) framework, explicitly activate or deactivate the constraint, which would otherwise only happen by chance or by propagation. Suitable splitting points for $x$ are multiples of $\pi$, since these can restrict the cosine to monotone branches, thereby enabling stronger deductions through satisfaction of the application conditions of the respective deduction rules (see page 105).

- *Bounded Cone of Influence (BCOI) reduction.* BCOI reduction, introduced for propositional BMC formulae in [21], is a technique which reduces the size of a BMC formula by removing variables and constraints which cannot affect the valuation of the variables occuring in the property expression. BCOI reduction is based on a simple syntactical analysis of the BMC formula and in general reduces the solving time. It should be performed as a standard preprocessing step when applying HYSAT-2 to BMC problems.

- *Support for optimization problems.* In many practical applications it is not only required to know whether a formula has a solution or not, but also to find a solution which is optimal w.r.t. some cost function. One might e.g. be interested in finding a mapping of tasks to processor nodes which minimizes the number of communications on the bus, or in determining an execution of a hybrid system which minimizes the consumption of some critical resource. Such problems can be solved by calling ISAT repeatedly within a binary search scheme which incrementally prunes the range of the auxiliary variable representing the value of the cost function in ISAT until the optimal value is attained with sufficient accuracy. Since ISAT's input formula may be built from Boolean, integer, and

real-valued variables using all standard Boolean and arithmetic operations, the class of optimization problems which can be tackled this way is very general and in particular includes nonlinear, nonconvex, and mixed-integer problems.

*Linear Relaxations of Nonlinear Constraints:* PSAT

The general principle underlying ISAT is to apply branching and deduction in order to reduce the complex problem of deciding the satisfiability of arithmetic constraint formulae with arbitrary Boolean structure to a sequence of very simple problems, each consisting in deciding the satisfiability of a conjunction of bound constraints. Bounds are the basic objects manipulated by ISAT. They are dynamically created by branching and by deduction, and — being input and output of deduction rules — are the nodes of the implication graph. At any time, the conjunction of all asserted bounds defines the portion of the search space which is currently investigated by the algorithm.

The choice of bounds as atoms of logical reasoning arose naturally from the attempt to integrate the DPLL procedure with interval constraint solving, because bounds allow to express both, interval borders and Boolean literals. This notwithstanding, it is certainly possible (and would be highly interesting) to build an ISAT solver which is based on a more general class of atomic constraints. A natural choice would be linear constraints, because satisfiability of conjunctions of linear constraints can be efficiently decided using linear programming. In the resulting solver — let us call it PSAT for the time being — linear constraints would take exactly the same role as bounds do in ISAT. While ISAT encloses potential solutions sets in products of *intervals*, the latter being defined by the set of asserted bounds, PSAT would enclose them in *polyhedra*, i.e. intersections of the halfspaces defined by the set of asserted linear constraints. Instead of narrowing bounds, PSAT's deduction rules would tighten linear relaxations of solution sets of arithmetic definitions, i.e. they would compute (locally, for each individual definition) linear inequalities which enclose the graph of the respective definition from above and from below. By this, they would, in general, achieve much tighter enclosings than the ones which could be obtained by using bound constraints (see figure 6.1). Linear programming would then be used to check the system of asserted linear constraints for consistency, and, in case of unsatisfiability, to detect a minimal reason for the conflict.

Computing linear relaxations of nonlinear constraints is a well-known technique in the field of nonlinear optimization [94, 115]. It seamlessly integrates with the ISAT framework, and we believe that ISAT's performance would considerably benefit from

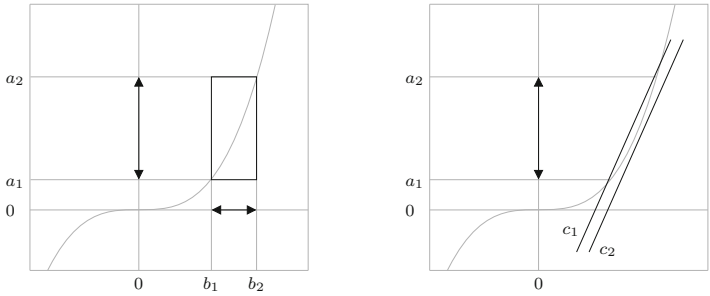


Figure 6.1: Given a definition $A = B^n$ with odd $n$ and an interval $[a_1, a_2]$ for $A$, bound deduction can enclose the graph of the definition only very coarsely in a box (left). Compared to this, the volume enclosed by the polyhedron built from bounds $a_1$ and $a_2$ and the linear constraints $c_1$ and $c_2$, which have been deduced from the bounds, is much smaller (right).

the tighter enclosures obtained by the use of linear relaxations. We consider the pSAT algorithm sketched above to be the most natural integration of linear programming into iSAT.

*Interval Newton Method for Certification of Candidate Solutions*

The Interval Newton method is the interval analog of the classical Newton method for finding roots of a continuously differentiable function $f : \mathbb{R} \to \mathbb{R}$. The method aims at iteratively contracting an initial interval $I_0$ such that possible roots of $f$ within $I_0$ are not removed. This is achieved by computing a sequence $I_{k+1} = I_k \cap N_f(I_k)$, where $N_f$ is the so-called Newton operator. The latter is defined as $N_f(I) = x - f(x)/F'(I)$, where $x$ is some point from $I$, usually its midpoint, and $F'(I)$ is the range of $f$'s first derivative $f'$ on $I$, which is computed using interval arithmetic. If $0 \in F'(I)$, the Newton operator is undefined. In this case $I$ can be split into subintervals on which $f'$ has no zeros, which are then processed by the Interval Newton method recursively. Remarkably, the following holds:

1. If $N_f(I_k) \cap I_k = \emptyset$ for some $k \in \mathbb{N}$, then $f$ has no roots in $I_0$.
2. If $N_f(I_k) \subseteq I_k$ for some $k \in \mathbb{N}$, then $f$ has a root in $I_k$.

It would be interesting to investigate to what extent this result, which in the same form holds for the multivariate Interval Newton method for multiple constraints (cf. [17],

page 585f), is useful to certify or refute the existence of solutions within candidate solution boxes computed by iSAT. It may certainly happen that neither (1) nor (2) applies, e.g. if the iteration fails to converge. Due to the undecidability of the formula class we are dealing with, this has to be necessarily so. On the other hand, it is well-known that the convergence of Newton's method is particularly good if the starting point of the iteration is close to the root. Given that the candidate solution boxes delivered by iSAT should provide excellent enclosures of potential roots, there is hope that in practice convergence occurs in many cases.

*Propagators for Differential Equations*

In section 2.2.2 we have explained hybridization as a method which transforms a hybrid automaton into another one which safely approximates the original continuous dynamics with simpler dynamics. This is achieved by

a) splitting control modes of the original automaton into submodes, where each submode is assigned to a certain part of the state space (a so-called *cell*), and

b) replacing, within each submode $v$, the flow predicate $p$ of the original control mode with a weaker predicate $p'$, such that $p$ conjoined with the contraints defining $v$'s cell implies $p'$.

Obviously, hybridization is a split-and-deduce method, just like the one employed in iSAT, performed on the automaton level, however. The interesting point here is, that the deduction mechanism directly deals with differential equations, not only with conventional arithmetic operations, as interval constraint solving and iSAT do. Deduction derives, after each splitting step performed, new differential (in)equations whose solution curves enclose the solution(s) of the original flow constraint.

This suggests to incorporate the same form of deduction directly into iSAT, and by this enable iSAT to natively handle differential equations. To this end, iSAT would support, besides ordinary arithmetic definitions, also definitions which contain dotted variables, like $\dot{x} = x - x^2$, for example.[4] As usual when dealing with differential equations, the $x$ occuring in $\dot{x} = x - x^2$ does *not* represent a variable, but a time-dependent function $x(t)$, which is a solution of the differential equation. Likewise, $\dot{x}$ denotes a function over $t$, namely the first derivative of $x(t)$. Narrowing the interval of $x$ by splitting or deduction thus means that the range of $x(t)$ is restricted accordingly for the entire duration of the flow, i.e. $x(t)$ is confined to a box (or *cell*) in the state

[4] The right-hand side $x - x^2$ could be decomposed into further definitions.

space whose upper and lower bounds are given by the interval borders of $x$. Similarly, bounds on $\dot{x}$ apply to the entire flow. iSAT's deduction rules would, however, treat $\dot{x}$ and $x$ simply as names of interval-valued variables, and, given bounds on $x$, deduce new bounds on $\dot{x}$ and vice versa. As illustrated in figure 6.2 a), interval propagation yields $\left(x \geq \frac{1}{2}, x \leq 1\right) \xrightarrow{\dot{x} = x - x^2} \left(\dot{x} \geq 0, \dot{x} \leq \frac{1}{4}\right)$, for example. The bounds deduced for $\dot{x}$ are interpreted by the component of iSAT which checks all asserted atoms for consistency. To this end, it replaces bounds on $\dot{x}$ with their corresponding closed-form solutions, the latter being linear inequalities, relating inital and final values of the flow. Figure 6.2 b) depicts the solution curves of $\dot{x} \geq 0$ and $\dot{x} \leq \frac{1}{4}$, which enclose the actual solution of $\dot{x} = x - x^2$ (printed as dashed line), where $x(0) = \frac{1}{2}$ has been chosen as initial condition. Since bounds on dotted variables have linear solution functions, this method blends best with the pSAT solver proposed in the previous section, because pSAT features a linear solver for checking the consistency of asserted atoms.

Given a consistency checker for constraints involving the exponential function, we could carry the approach one step further and deduce linear differential inequalities (instead of bounds) which are locally implied by a differential equation. In above example, we could deduce $\left(x \geq \frac{1}{2}, x \leq 1\right) \xrightarrow{\dot{x} = x - x^2} \left(\dot{x} \geq -\frac{1}{2}x + \frac{1}{2},\ \dot{x} \leq -\frac{1}{2}x + \frac{9}{16}\right)$ for instance, as illustrated in figure 6.2 c). As can be seen from figure 6.2 d), the nonlinear enclosure defined by the deduced linear differential inequations is much tighter than the linear envelope obtained from deduction of bounds on $\dot{x}$.

In general, contraction of $x$'s interval will restrict the possible duration of the flow, which has to end before it leaves its cell. In order to capture as much of the flow as possible within a single BMC step, it is therefore desirable to keep the interval of $x$ as wide as possible. On the other hand, the width of $x$'s interval must be small enough to yield a sufficiently accurate approximation of the flow. This can, for example, be achieved by adding constraints which ensure that at the beginning and at the end of the flow the difference between upper and lower boundary curve, i.e. the maximum approximation error, does not exceed a user-specified limit. These constraints will in turn entail a bound on the width of $x$'s interval and thus on the duration of the flow. If the flow, with these restrictions, cannot connect its potential initial and target states (i.e. the BMC formula is unsatisfiable), then the solver will be forced to unwind the BMC formula one step further and thus to split the flow into two fragments, each of which confined to its individual cell. Increasing the required accuracy thus yields error traces of increasing length, because flows have to be divided into tiny segments in order to keep the approximation error small.

According to the above description, deduction rules compute differential inequations which are implied by the current proof state and whose solution functions are generated by the solver backend which performs the consistency check. Alternatively, it would certainly be possible to build the deduction rules in such a way that they directly assert the boundary functions enclosing the flow, i.e. propagate

$$\left(x \geq \tfrac{1}{2}, x \leq 1\right) \xrightarrow{\dot{x} = x - x^2} \left(x(t) \geq x(0),\ x(t) \leq x(0) + \tfrac{1}{4}t\right),$$

where $x(0)$ and $x(t)$ are initial and final value of the flow, instead of

$$\left(x \geq \tfrac{1}{2}, x \leq 1\right) \xrightarrow{\dot{x} = x - x^2} \left(\dot{x} \geq 0, \dot{x} \leq \tfrac{1}{4}\right).$$

Moreover, the boundary functions need not necessarily be solutions of differential inequations derived in a previous step, but could as well be Taylor expansions, like those employed in section 2.2.3.

*Parallelization of* iSAT *via Partitioning of the Implication Queue*

It is well-known that the major part of the runtime of a propositional SAT solver (more than 90% according to [91]) is consumed by deduction. This is even more true for iSAT, because real-valued variables may be assigned many times during iSAT's deduction phase, not just once, like Boolean variables. Optimizations of the deduction routine therefore have a strong impact on the overall performance of the solver.

A central data structure used to implement deduction in modern SAT engines is the *implication queue* which stores all recent assignments whose deductive consequences have not been explored yet. Deduction rules add asserted atoms (i.e. literals in a SAT solver and bounds in iSAT) to the rear of the queue, while the inference engine removes them from the front and checks (through application of deduction rules) whether they yield further propagations. Deduction terminates if the implication queue runs empty.

Given that most CPUs today feature a multi-core architecture, a reasonable approach to speed up deduction would be to parallelize processing of the deduction queue by partitioning the latter into multiple subqueues, where the entries of each subqueue are processed by a separate process or thread. In particular, this would enable to run multiple specialized inference engines in parallel, e.g. one optimized for purely Boolean deduction, another one for ICP-based deduction, and a third one for LP-based reasoning. Each engine would be allocated on a different processor core and consume inputs from a dedicated subqueue, containing entries of the sort processed by the respective inference engine only.

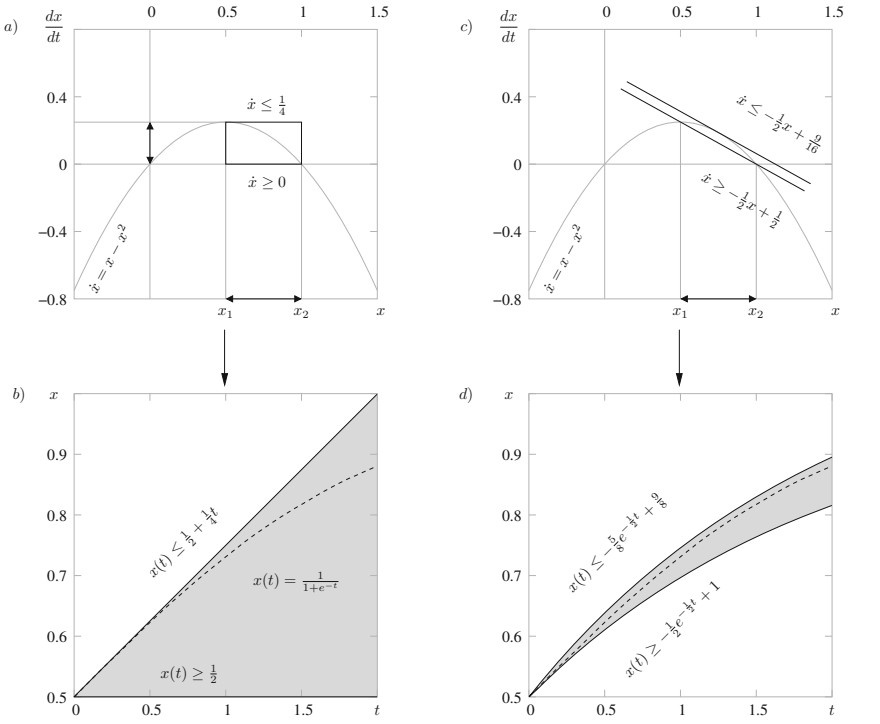


Figure 6.2: Deduction of differential inequations.

A common problem observed in solvers which combine different solving technologies, e.g. in DPLL($T$) solvers, is the high runtime penalty incurred through cache invalidation upon each handover between the different engines. Provided that the individual processor cores are equipped with caches of sufficient size, the form of parallelization proposed above would provide a neat solution to this problem, yielding accelerations which go beyond the speed-ups achieved by parallel processing of implication queue entries.

## 6.3 Final Thoughts

The work on proof engines for the analysis of hybrid systems is an appealing field of research: On the one hand it touches problems, e.g. questions concerning complexity

and decidability, which lie at the heart of the theory of computation, and on the other hand the resulting tools are of immediate practical use, as witnessed by the spectrum of applications quoted in section 6.1.

Modern SAT and SMT solvers, like those we dealt with in this thesis, are highly optimized, complex pieces of software. Their performance not only rests upon theoretical insights, but a good deal of it is based on clever implementation techniques and low-level code optimizations which, unfortunately, hardly make their way into publications. The combination of theoretical work, good programming skills and practical experience, which is indispensible for building efficient solvers, renders their development a challenging and interesting task.

A rule of thumb quoted in many books on algorithms states that NP-completeness marks the borderline between problems that can be efficiently solved and problems which are computationally intractable. According to this rule, none of the problems tackled in this thesis has an efficient solution — seemingly contradicting the title of this work. This notwithstanding, we have demonstrated that our tools can handle formulae with several ten-thousands of Boolean and continuous variables, even such involving nonlinear arithmetic constraints.

Yet, the problems we are dealing with are inherently hard, as witnessed by the fact that once in a while one discovers an innocent looking problem which has extremely long runtimes. An analysis of the reasons for such behaviour is usually intricate and tedious, but occasionally yields insights which help to improve the solving algorithms. Breakthroughs are rare, however, and one has to be prepared that a lot of work might yield very little increase in performance only. If at all. In fact, the work in a domain, where simple code optimization may yield bigger performance gains than sophisticated theoretical work, can also be quite frustrating. Still, this kind of work is mandatory to push technological development, even if progress comes in small steps only, which is the case in propositional SAT solving for some years now.

It took more than 30 years from the invention of the Davis-Putnam algorithm to the maturity of SAT solving for industrial applications. As to solvers for mixed propositional-numerical formulae, we are just at the beginning and will hopefully witness a similarly impressive development like in propositional SAT solving within the next decades.

# Bibliography


[1] Erika Ábrahám, Bernd Becker, Felix Klaedtke, and Martin Steffen. Optimizing bounded model checking for linear hybrid systems. In *Proceedings of VMCAI'05 (Verification, Model Checking, and Abstraction)*, volume 3385 of *Lecture Notes in Computer Science*, pages 396–412, Paris, January 2005. Springer-Verlag.

[2] Erika Ábrahám, Marc Herbstritt, Bernd Becker, and Martin Steffen. Memory-aware bounded model checking for linear hybrid systems. In Bernd Straube, editor, *ITG/GI/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, pages 153–162, February 2006. ISBN 3-9810287-1-6.

[3] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ILP versus specialized 0-1 ILP: An update. In *Proceedings of the ACM/IEEE Intl. Conf. Comp.-Aided Design (ICCAD)*, pages 450–457, November 2002.

[4] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. PBS: A backtrack search pseudo–Boolean solver. In *Proceedings of the International Symposium on the theory and applications of satisfiability testing (SAT)*, pages 346–353, 2002.

[5] Eugene Asarin, Thao Dang, Goran Frehse, Antoine Girard, Colas Le Guernic, and Oded Maler. Recent progress in continuous and hybrid reachability analysis. In *Proceedings of the IEEE International Symposium on Computer-Aided Control Systems Design*, pages 1582–1587, Munich, Germany, 2006.

[6] Eugene Asarin, Thao Dang, and Antoine Girard. Hybridization methods for the analysis of nonlinear systems. *Acta Informatica*, 43(7):451–476, 2007.

[7] Gilles Audemard, Marco Bozzano, Alessandro Cimatti, and Roberto Sebastiani. Verifying industrial hybrid systems with MathSAT. *ENTCS*, 89(4), 2004.



C. Herde, *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure*, DOI 10.1007/978-3-8348-9949-1,

[8] Andrea Balluchi, Luca Benvenuti, Maria Domenica Di Benedetto, Tiziano Villa, and Alberto L. Sangiovanni-Vincentelli. Idle speed control: A benchmark for hybrid system research. In *Proceedings of the 2nd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'06)*, Alghero, Italy, 2006.

[9] Luís Baptista and João P. Marques Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, 2000.

[10] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on Demand in SAT Modulo Theories. In M. Hermann and A. Voronkov, editors, *13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'06*, volume 4246 of *LNCS*, pages 512–526. Springer, 2006.

[11] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[12] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Biere et al. [22], pages 825–885.

[13] Peter Barth. A Davis–Putnam based enumeration algorithm for linear pseudo–Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1995.

[14] Andreas Bauer, Markus Pister, and Michael Tautschnig. Tool-support for the analysis of hybrid systems and models. In *Proceedings of the 2007 Conference on Design, Automation and Test in Europe (DATE'07)*, Los Alamitos, CA, April 2007. IEEE Computer Society.

[15] Alberto Bemporad and Manfred Morari. Verification of hybrid systems via mathematical programming. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control (HSCC'99)*, volume 1569 of *Lecture Notes in Computer Science*, pages 31–45. Springer-Verlag, 1999.

[16] Frédéric Benhamou. Heterogeneous constraint solving. In *Proceedings of the 5th International Conf. on Algebraic and Logic Programming*, volume 1139 of *LNCS*. Springer, 1996.

[17] Frédéric Benhamou and Laurent Granvilliers. Continuous and interval constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 16. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006.

[18] Frédéric Benhamou, David A. McAllester, and Pascal Van Hentenryck. CLP (intervals) revisited. In *Proceedings of the 1994 International Symposium on Logic programming (ILPS'94)*, pages 124–138, Cambridge, MA, USA, 1994. MIT Press.

[19] Gérard Berry, Hubert Comon, and Alain Finkel, editors. *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*. Springer, 2001.

[20] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[21] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. Verifying safety properties of a powerPC microprocessor using symbolic model checking without BDDs. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, pages 60–71. Springer-Verlag, 1999.

[22] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[23] Aart J. C. Bik and Harry A. G. Wijshoff. Implementation of Fourier-Motzkin elimination. Technical Report TR94-42, Dpt. of Computer Science, University of Leiden, The Netherlands, 1994.

[24] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Proceedings of the 3rd International Conference on Formal Meth-*

*ods in Computer-Aided Design (FMCAD'00)*, pages 372–389, London, UK, 2000. Springer-Verlag.

[25] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In Berry et al. [19], pages 454–464.

[26] Alexander Bockmayr and Volker Weispfenning. Solving numerical constraints. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 12, pages 751–842. Elsevier and MIT Press, Amsterdam, the Netherlands, 2001.

[27] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic SMT solver. In *Proceedings of the 20th international conference on Computer Aided Verification (CAV'08)*, pages 294–298, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Antero Junttila, Rossum Peter van, Stephan Peter Schulz, and Roberto Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, pages 317–333. Springer, 2005.

[29] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[30] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Deciding CLU logic formulas via Boolean and pseudo–Boolean encodings. In *In Proceedings of the International Workshop on Constraints in Formal Verification (CFV'02)*, 2002.

[31] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 428–439, Philadelphia, Pennsylvania, USA, 1990. IEEE Computer Society.

[32] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *Proceedings of the 40th Design Automation Conference (DAC'03)*, pages 830–835, Anaheim (California, USA), June 2003. ACM.

[33] John W. Chinneck. Finding a useful subset of constraints for analysis in an infeasible linear program. *INFORMS Journal on Computing*, 9(2):164–174, 1997.

[34] John W. Chinneck and E. W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.

[35] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Computational challenges in bounded model checking. *STTT*, 7(2):174–183, 2005.

[36] John G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[37] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Barkhage, editor, *Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer, 1975.

[38] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd annual ACM symposium on Theory of computing (STOC'71)*, pages 151–158, New York, NY, USA, 1971. ACM.

[39] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In Berry et al. [19], pages 436–453.

[40] George Bernhard Dantzig. Maximization of a linear function of variables subject to linear inequalities. In T. C. Koopmans, editor, *Activity Analysis of Production and Allocation - Proceedings of a Conference*, volume 13 of *Cowles Commission Monograph*, pages 339–347. Wiley, New York, 1951.

[41] Ernest Davis. Constraint propagation with interval labels. *Artif. Intell.*, 32(3):281–331, 1987.

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

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

[44] Thierry Boy de la Tour. An optimality result for clause form translation. *J. Symb. Comput.*, 14(4):283–301, 1992.

[45] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[46] Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222, Cork, Ireland, July 2004. Springer-Verlag.

[47] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[48] Marcel Dhiflaoui, Stefan Funke, Carsten Kwappik, Kurt Mehlhorn, Michael Seel, Elmar Schömer, Ralph Schulte, and Dennis Weber. Certifying and repairing solutions to large LPs how good are LP-solvers? In *SODA '03: Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms*, pages 255–256, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[49] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[50] Bruno Dutertre and Leonardo de Moura. Integrating simplex with DPLL(T). Technical Report SRI-CSL-06-01, SRI International, 2006.

[51] Niklas Eén and Niklas Sörensson. Translating pseudo–Boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[52] Andreas Eggers. Induktive Verifikation linearer Hybrider Systeme. BSc thesis, Carl von Ossietzky Universität, Department of Computing Science, Oldenburg, Germany, 2005.

[53] Andreas Eggers, Natalia Kalinnik, Stefan Kupferschmid, and Tino Teige. Challenges in constraint-based analysis of hybrid systems. In Angelo Oddi, François Fages, and Francesca Rossi, editors, *Recent Advances in Constraints – 13th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2008, Rome, Italy, June 18-20, 2008, Revised Selected Papers*, volume 5655 of *Lecture Notes in Artificial Intelligence*, pages 51–65, Berlin, Heidelberg, 2009. Springer.

[54] Sebastian Engell, Stefan Kowalewski, Christian Schulz, and Olaf Stursberg. Continuous–discrete interactions in chemical processing plants. *Proceedings of the IEEE*, 88(7):1050–1068, July 2000.

[55] Jacob Enslev, Anne-Sofie Nielsen, Martin Fränzle, and Michael R. Hansen. Bounded model construction for duration calculus. In Neil Jones et al., editor, *Proceedings of the 17th Nordic Workshop on Programming Theory (NWPT 05)*. Københavns Universitet, October 2005.

[56] Julius Farkas. Theorie der einfachen Ungleichungen. *Journal für die reine und angewandte Mathematik*, 124:1–27, 1901.

[57] Germain Faure, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez Carbonell. SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers. In Hans Kleine Buning and Xishun Zhao, editors, *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing, SAT'08*, volume 4996 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2008.

[58] Jean Baptiste Joseph Fourier. Solution d'une question particulière du calcul des inégalités. *Nouveau Bulletin des Sciences par la Société Philomatique de Paris*, pages 99–100, 1826.

[59] Martin Fränzle. Take it NP-easy: Bounded model construction for duration calculus. In Ernst-Rüdiger Olderog and Werner Damm, editors, *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant systems (FTRTFT 2002)*, volume 2469 of *Lecture Notes in Computer Science*, pages 245–264. Springer-Verlag, 2002.

[60] Martin Fränzle and Christian Herde. Efficient SAT engines for concise logics: Accelerating proof search for zero-one linear constraint systems. In Andrei

Voronkov and Moshe Y. Vardi, editors, *Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2003)*, volume 2850 of *LNCS, subseries LNAI*, pages 302–316. Springer Verlag, 2003.

[61] Martin Fränzle and Christian Herde. Efficient proof engines for bounded model checking of hybrid systems. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, 2004.

[62] Martin Fränzle and Christian Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30:179–198, 2007.

[63] Martin Fränzle, Christian Herde, Stefan Ratschan, Tobias Schubert, and Tino Teige. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *JSAT Special Issue on Constraint Programming and SAT*, 1:209–236, 2007.

[64] Goran Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *Int. J. Softw. Tools Technol. Transf.*, 10(3):263–279, 2008.

[65] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[66] John Gleeson and Jennifer Ryan. Identifying minimally infeasible subsystems of inequalities. *INFORMS Journal on Computing*, 2(1):61–63, 1990.

[67] Jan Frisco Groote, Wilco Koorn, and Sebastian van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *Proceedings of the 10th Annual Conference on Computer Assurance (Compass'95)*, pages 57–68, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.

[68] Jan Frisco Groote and Joost P. Warners. The propositional formula checker HeerHugo. Technical report SEN-R9905, CWI, 1999.

[69] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1996.

[70] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill Inc., 1998.

[71] Eric C. R. Hehner. Predicative programming. *Communications of the ACM*, 27:134–151, 1984.

[72] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

[73] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *Proceedings of the 27th annual ACM symposium on Theory of computing (STOC'95)*, pages 373–382, New York, USA, 1995. ACM.

[74] Christian Herde, Andreas Eggers, Martin Fränzle, and Tino Teige. Analysis of hybrid systems using HySAT. In *Proceedings of the 3rd International Conference on Systems (ICONS'08)*, pages 196–201, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[75] Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. Interval arithmetic: from principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

[76] Paul Jackson and Daniel Sheridan. Clause form conversions for Boolean circuits. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers)*, volume 3542 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2004.

[77] Paul Jackson and Daniel Sheridan. The optimality of a fast CNF conversion and its use with SAT. Technical Report APES-82-2004, APES Research Group, March 2004.

[78] Hoonsang Jin and Fabio Somenzi. An incremental algorithm to check satisfiability for bounded model checking. *ENTCS*, 119, 2004.

[79] Narenda Jussien and Olivier Lhomme. Dynamic domain splitting for numeric CSPs. In *European Conference on Artificial Intelligence*, pages 224–228, 1998.

[80] Leonid Genrikhovich Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[81] Adam B. Kinsman and Nicola Nicolici. Finite precision bit–width allocation using SAT–modulo theory. In *Proceedings of the 2009 Conference on Design, Automation and Test in Europe (DATE'09)*, pages 1106–1111, 2009.

[82] Stefan Kowalewski and Olaf Stursberg. The batch evaporator: A benchmark example for safety analysis of processing systems under logic control. In *Proceedings 4th Workshop on Discrete Event Systems (WODES'98)*, pages 302–307, London, 1998.

[83] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[84] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. Circuit-based boolean reasoning. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 232–237, New York, NY, USA, 2001. ACM.

[85] Inês Lynce, Luís Baptista, and João P. Marques Silva. Complete search restart strategies for satisfiability. In *Proceedings of the IJCAI'01 Workshop on Stochastic Search Algorithms (IJCAI-SSA)*, August 2001.

[86] Vasco M. Manquinho and Olivier Roussel. The first evaluation of pseudo–Boolean solvers (PB'05). *Journal on Satisfiability, Boolean Modeling and Computation(JSAT)*, 2:103–143, 2006.

[87] Vasco M. Manquinho and João P. Marques Silva. On using cutting planes in pseudo–Boolean optimization. *JSAT*, 2(1-4):209–219, 2006.

[88] Yuri Matiyasevich. *Hilbert's Tenth Problem*. MIT Press, Cambridge, Massachusetts, 1993.

[89] Kenneth L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[90] Ramon Edgar Moore. *Interval Analysis*. Prentice Hall, NJ, 1966.

[91] 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)*, 2001.

[92] Theodore Samuel Motzkin. *Beiträge zur Theorie der linearen Ungleichungen*. Doctoral dissertation, Universität Zürich, 1936.

[93] Arnold Neumaier. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, Cambridge, 1990.

[94] Arnold Neumaier. Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica*, 13:271–369, 2003.

[95] Arnold Neumaier and Oleg Shcherbina. Safe bounds in linear and mixed-integer linear programming. *Math. Program.*, 99(2):283–296, 2004.

[96] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 6, pages 335–367. Elsevier, Amsterdam, Netherlands, 2001.

[97] Sören Ollhoff. Automatische Übersetzung von Simulink Modellen in HySAT-Formeln. BSc thesis, Carl von Ossietzky Universität, Department of Computing Science, Oldenburg, Germany, 2007.

[98] Jonathan H. Owen and Sanjay Mehrotra. On the value of binary expansions for general mixed-integer linear programs. *Operations Research*, 50(5):810–819, 2002.

[99] Viresh Paruthi and Andreas Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. *Computer Design, International Conference on*, 0:459, 2000.

[100] Marc E. Pfetsch. *The Maximum Feasible Subsystem Problem and Vertex-Facet Incidences of Polyhedra*. Doctoral dissertation, TU Berlin, 2002.

[101] Florian Pigorsch, Christoph Scholl, and Stefan Disch. Advanced unbounded model checking based on AIGs, BDD sweeping, and quantifier scheduling. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)*, pages 89–96, Washington, DC, USA, 2006. IEEE Computer Society.

[102] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.

[103] Steven David Prestwich. CNF encodings. In Biere et al. [22], pages 75–97.

[104] Stefan Ratschan. Efficient solving of quantified inequality constraints over the real numbers. *ACM Transactions on Computational Logic*, 7(4):723–748, 2006.

[105] Andreas Schäfer and Mathias John. Conceptional modeling and analysis of spatio-temporal processes in biomolecular systems. In Sebastian Link and Markus Kirchberg, editors, *Proceedings of the 6th Asia-Pacific Conference on Conceptual Modelling (APCCM 2009)*, volume 96 of *CRPIT*, pages 39–48, Wellington, New Zealand, 2009. ACS.

[106] Carsten Schild. Anwendung induktiver Verifikation mittels DPLL–basierten arithmetischen Constraint–Solvings auf eine ETCS–Fallstudie. Student research project, Carl von Ossietzky Universität, Department of Computing Science, Oldenburg, Germany, 2009.

[107] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[108] Roberto Sebastiani. Lazy satisfiability modulo theories. *JSAT Special Issue on Satisfiability Modulo Theories*, 3:141–224, 2007.

[109] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, pages 108–125, London, UK, 2000. Springer-Verlag.

[110] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo–Boolean SAT solver. *JSAT*, 2(1-4):165–189, 2006.

[111] João P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, September 1999.

[112] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[113] Ofer Strichman. Tuning SAT checkers for bounded model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, pages 480–494, London, UK, 2000. Springer-Verlag.

[114] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkely, California, 1951. Previous version published as a technical report by the RAND Corporation, 1948.

[115] Mohit Tawarmalani and Nikolaos V. Sahinidis. Global optimization of mixed-integer nonlinear programs: A theoretical and computational study. *Math. Program.*, 99(3):563–591, 2004.

[116] Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2009.

[117] Fabio Danilo Torrisi. *Modeling and Reach-Set Computation for Analysis and Optimal Control of Discrete Hybrid Automata*. PhD thesis, ETH Zrich, 2003.

[118] Gregory S. Tseitin. On the complexity of derivations in propositional calculus. In A. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logics*, 1968.

[119] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, pages 25–57, 2006.

[120] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 542–545, Las Vegas (Nevada, USA), June 2001.

[121] Steven A. Wolfman and Daniel S. Weld. The LPSAT engine & its application to resource planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 310–316, Stockholm, Sweden, 1999.

[122] Mohamed H. Zaki, Ian M. Mitchell, and Mark M. Greenstreet. DC operating point analyis: a formal approach. Workshop on Formal Verification of Analog Circuits (FAC'09), 2009.

[123] 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. Springer, 1997.

[124] Hantao Zhang and Mark E. Stickel. An efficient algorithm for unit-propagation. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, pages 166–169, Fort Lauderdale (Florida USA), 1996.

[125] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, November 2001.

[126] Lintao Zhang and Sharad Malik. The quest for efficient Boolean satisfiability solvers. In *Proceedings of the 18th International Conference on Automated Deduction (CADE'02)*, pages 295–313, London, UK, 2002. Springer-Verlag.

# Index




C. Herde, *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure*, DOI 10.1007/978-3-8348-9949-1,