PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

An Effective SMT Engine for Formal Verification

Alberto Griggio

Advisor:

Prof. Roberto Sebastiani

DISI - University of Trento

Co-Advisor:

Dr. Alessandro Cimatti

FBK-IRST, Trento

December 2009

This PhD program has been partly supported by Semiconductor Research Corporation (SRC) under Global Research Collaboration (GRC) Custom Research Project 2009-TJ-1880 WOLFLING.

Abstract

Formal methods are becoming increasingly important for debugging and verifying hardware and software systems, whose current complexity makes the traditional approaches based on testing increasingly-less adequate. One of the most promising research directions in formal verification is based on the exploitation of Satisfiability Modulo Theories (SMT) solvers. In this thesis, we present MathSAT, *a modern, efficient SMT solver that provides several important functionalities, and can be used as a workhorse engine in formal verification. We develop novel algorithms for two functionalities which are very important in verification – the extraction of unsatisfiable cores and the generation of Craig interpolants in SMT – that significantly advance the state of the art, taking full advantage of modern SMT techniques. Moreover, in order to demonstrate the usefulness and potential of SMT in verification, we develop a novel technique for software model checking, that fully exploits the power and functionalities of the SMT engine, showing that this leads to significant improvements in performance.*

Keywords

[SMT, Formal Verification, Craig Interpolants, Unsatisfiable Cores, Software Model Checking, Theorem Proving, Automated Deduction]

Contents

List of Tables

List of Figures

Introduction

The progress in the fields of electronics and computer science has led to the realization of extremely sophisticated hardware and software systems, which are capable of performing very complex tasks. Nowadays, such systems are ubiquitous in human activities, and they are a fundamental component of applications in a multitude of critical sectors. Therefore, it is extremely important to ensure that they operate correctly.

The traditional approaches to assess the correctness of a hardware/software system are mainly based on *simulation* and *testing* techniques: the system (and its sub-components) is executed on a series of representative inputs, in order to check that the behaviour is the expected one. The effectiveness of such techniques is directly proportional to the degree of coverage that they are able to ensure. With the increasing complexity of current systems, however, ensuring a good coverage is becoming more and more difficult.

An increasingly-popular approach for tackling this problem is that of complementing the traditional techniques with verification techniques based on *formal methods*. Such techniques combine the rigor of mathematical methods and the availability of efficient computer programs in order to produce a demonstration of the correctness of a system with respect to a specification written in some formal mathematical language. Examples of formal verification techniques include *model checking*, *equivalence check-*

ing, *symbolic simulation*, *static analysis*, *abstract interpretation*. The last twenty years have witnessed an impressive progress in such techniques, which are nowadays routinely applied by companies such as Intel, IBM, Microsoft.

The foundations of formal verification lie in *mathematical logic*. Logical formulae are used to formally specify systems, their behaviours, and the properties that they should satisfy. Therefore, computer programs and algorithms that are able to manipulate logical formulae – such as *theorem provers*, *decision diagrams*, *quantifier elimination and Craig interpolation procedures* – are a key component of formal verification techniques.

Such tools and algorithms typically present a trade-off between the expressiveness of the logic that they can handle on the one hand, and the efficiency, scalability and degree of automation that they can ensure on the other hand. For such reason, the simple propositional logic is used extensively in formal verification, thanks to the availability of very efficient decision procedures for it, such as *binary decision diagrams (BDDs)* and *SAT solvers*. The latter in particular have seen tremendous improvements in the last fifteen years, making verification techniques based on SAT very successful, especially in the context of hardware systems. An important factor for this success is that modern SAT solvers are not only capable of proving efficiently the satisfiability of huge propositional formulae, but they also provide several other functionalities, such as model generation and enumeration, proof production, extraction of unsatisfiable cores, generation of (Craig) interpolants, that have been exploited successfully in a number of verification techniques.

The formalism of plain propositional logic, however, is often not suitable or expressive enough for representing many real-world problems, including the verification of hardware designs at the Register Transfer Level (RTL), of real-time and hybrid control systems, and the analysis of proof obligations

in software verification. Such problems are more naturally expressible as satisfiability problems in decidable first-order theories. For example, RTL designs can be formalized using a combination of the theory of *bit-vectors* and the theory of *equality and uninterpreted functions*; the theory of *arithmetic* (both linear and non-linear) over the reals is often used for real-time and hybrid systems verification; for software verification, combinations of bit-vectors or linear arithmetic on the integers with uninterpreted functions and *theories for data structures* (e.g. arrays, lists, sets) are usually adopted. The need for a higher degree of expressiveness moved the attention of the scientific and industrial community towards a new generation of theorem provers, that combine the efficiency of modern SAT solvers with the ability of reasoning about decidable theories. This new paradigm is known as *Satisfiability Modulo Theories (SMT)*.

Thanks to the huge progress of the last few years, SMT is now becoming a viable alternative to SAT for formal verification, promising to convey the same high levels of automation, efficiency and scalability, while offering a much higher expressive power. It is a widespread opinion in the verification community that an effective exploiting of SMT will be a key factor in the progress of formal verification.

SMT, however, is still a relatively new paradigm, and current SMT solvers still suffer from some limitations. In particular, the research on SMT has mainly focused on the problem of deciding the satisfiability of formulae, and on developing efficient decision procedures for several theories, reserving significantly less interest for other functionalities, such as generation of *proofs*, *simplification* of formulae, extraction of small *unsatisfiable cores*, computation of *interpolants*, which are however extremely useful in the context of formal verification.

In this thesis we address some of these limitations. We present MathSAT, a modern, efficient SMT solver, which is able to deal efficiently with

formulae expressed in combinations of several important theories. We describe the functionalities that MathSAT provides, focusing in particular on the extraction of small unsatisfiable cores and the efficient generation of interpolants. Moreover, we show how it is possible to significantly boost one well-known technique for software model checking by fully exploiting the power and the functionalities of a modern SMT solver like MathSAT.

In particular, we make the following contributions:

1. We present and discuss several procedures, techniques and implementation details of MathSAT, a state-of-the-art SMT solver. We focus in particular on aspects that are often omitted from research papers on SMT, but which can play a significant role in practice for performance.

2. We address for the first time the problem of computing small unsatisfiable cores in SMT, by presenting a novel, SMT-specific approach to it, which we call the *Lemma-Lifting approach*. An important feature of this approach is that it allows for exploiting for free all the techniques for the extraction of small unsatisfiable cores of propositional formulae, a problem well-studied in the SAT community, for which several very effective algorithms exist. We describe our algorithm, discuss its features, and empirically evaluate it to show its effectiveness and its efficiency.

3. We describe novel techniques for efficiently generating Craig interpolants for SMT problems, fully leveraging the algorithms used in a state-of-the-art SMT solver. We show how to extend efficient SMT solving techniques to SMT interpolation, for a wide class of important theories and their combinations, without paying a substantial price in performance. We give specialized algorithms for *linear arithmetic* over the rationals, *difference logic*, and the theory of *unit-two-variables-per-*

inequality ($\mathcal{UTVPI}$); moreover, we describe a general interpolation algorithm for *combinations of convex theories* based on the *Delayed Theory Combination (DTC)* technique. We present an interpolating SMT solver that is able to produce interpolants for a much wider class of problems than its competitors, and, on problems that can be dealt with by other tools, shows dramatic improvements in performance, often by orders of magnitude.

4. We propose a novel technique for software model checking in the context of counterexample-guided-abstraction-refinement (CEGAR) with lazy abstraction, which we call *Large-Block Encoding (LBE)*. The technique is a generalization of a successful approach to software model checking which we refer to as *Single-Block Encoding (SBE)*. LBE was specifically conceived to exploit better than SBE the power and functionalities of modern SMT solvers. We evaluate LBE on a standard set of benchmark C programs, and show that, by leveraging the efficiency of state-of-the-art SMT techniques, it outperforms the traditional SBE approach.

Structure of the thesis

This thesis is divided into three parts.

Part I is devoted to the description of the main components of a modern SMT solver, and of MathSAT in particular. Chapter 1 reviews theoretical results and algorithms at the basis of the lazy SMT approach. We give an overview of the state of the art in SMT solving, covering the algorithm for integrating a SAT solver with theory-specific decision procedures which underlies lazy SMT solvers, its main optimizations, decision procedures for the most frequently-used theories, and methods for theory combination. In chapter 2, we present the MathSAT SMT solver in more detail. The aim

of this description is to provide details that are often omitted from research papers on SMT, but which from our experience can play a significant role in practice for performance. We describe its architecture and we discuss its main design choices, implementation details and optimization techniques. Where appropriate, we also present experimental results demonstrating the usefulness of the optimization techniques described.

Part II is dedicated to the description of the extended functionalities provided by MathSAT, that go beyond simply checking the satisfiability of a formula. Chapter 3 deals with the extraction of unsatisfiable cores. We first review the state of the art in unsatisfiable core extraction for SAT and SMT formulae. We then introduce our novel Lemma-Lifting approach and discuss its features. Finally, we present an extensive empirical evaluation on a wide set of benchmarks, in order to compare the Lemma-Lifting approach to previous algorithms. We also analyze the impact of different configurations of our algorithm, in order to demonstrate its versatility. Chapter 4 describes our novel techniques for the generation of Craig interpolants in SMT. After reviewing the state of the art, we describe our novel interpolation algorithms for linear arithmetic over the rationals, difference logic, the $\mathcal{UTVPI}$ theory and theory combination using the DTC technique. We conclude the chapter with an experimental evaluation of our techniques, in which we compare their implementation within MathSAT with the other available tools, showing significant improvements in performance and scalability.

In part III we present an application of MathSAT to the formal verification of software. We present our new Large-Block Encoding technique in chapter 5, we compare it with the traditional Single-Block Encoding, and we show significant performance improvements on a standard set of benchmark C programs. We also provide the needed background on software model checking and discuss related work.

Finally, in chapter 6 we draw some conclusions and we outline possible directions for future research.

Note. *Although* MathSAT *fully supports the theory of* bit-vectors *($\mathcal{BV}$), in this thesis we shall not deal with it. In fact, almost all the work on $\mathcal{BV}$ in the current version of* MathSAT *was done by Anders Franzén, and a detailed description of the techniques used can be found in his Ph.D. thesis.*

Part I

MathSAT: an Efficient SMT Solver

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to (a decidable fragment of) some decidable first-order theory $\mathcal{T}$. SMT has important applications in several different domains, one of the most important being formal verification of both hardware and software systems. In the last few years, SMT has been exploited successfully for several verification tasks.

A key factor for the successful application of SMT is the availability of efficient decision procedures for it, called *SMT solvers*, supporting several expressive theories (e.g., bit-vectors, linear arithmetic, arrays) able to represent problems which are either not expressible by Boolean logic or which can be expressed to a much higher level of abstraction, and capable of scaling to very large and complex problems. Most modern SMT solvers are based on the *lazy* approach, in which a propositional SAT solver (based on the DPLL algorithm) is combined with $\mathcal{T}$-specific decision procedures for conjunctions of constraints in the theory $\mathcal{T}$.

In the first part of the thesis, we describe our tool MathSAT, one of the most efficient lazy SMT solvers available. MathSAT implements many of the state-of-the-art algorithms and techniques that have been proposed in the SMT community over the last few years, and it supports several important theories and combinations. Its efficiency is demonstrated by

the results of the last three editions of the annual SMT solvers competition SMT-COMP (`http://smtcomp.org`). In particular, it obtained the following results:

- In 2007, MathSAT 4.0 competed in 7 divisions, obtaining 2 third places;
- In 2008, MathSAT 4.2 competed in 9 divisions, obtaining 3 second places and 4 third places;
- In 2009, MathSAT 4.3 competed in 12 divisions, obtaining 2 first places, 7 second places and 1 third place.

In this part, we first review the state-of-the-art algorithms for quantifier-free lazy SMT solving on which MathSAT is based (Ch. 1), and we then describe the main aspects of its architecture and we provide details about the implementation of its main components (Ch. 2).

Chapter 1

Background

This chapter introduces background concepts and terminology that shall be used both in this part and in the rest of the thesis. The material presented is standard in SMT, and is mostly taken from [BCF+08, CGS09b, Seb07].

1.1 The Satisfiability Modulo Theory Problem

Our setting is standard first-order logic.

In the following, let Σ be a first-order signature containing function and predicate symbols with their arities, and $\mathcal{V}$ be a set of variables. A 0-ary function symbol c is called a constant. A 0-ary predicate symbol A is called a Boolean atom. A Σ-term is either a variable in $\mathcal{V}$ or it is built by applying function symbols in Σ to Σ-terms. If $t_1, \ldots, t_n$ are Σ-terms and P is a predicate symbol, then $P(t_1, \ldots, t_n)$ is a Σ-atom. A Σ-formula φ is built in the usual way out of the universal and existential quantifiers $\forall, \exists$, the Boolean connectives $\wedge, \neg$, and Σ-atoms. We use the standard Boolean abbreviations: "$\varphi_1 \vee \varphi_2$" for "$\neg(\neg\varphi_1 \wedge \neg\varphi_2)$", "$\varphi_1 \rightarrow \varphi_2$" for "$\neg(\varphi_1 \wedge \neg\varphi_2)$", "$\varphi_1 \leftrightarrow \varphi_2$" for "$\neg(\varphi_1 \wedge \neg\varphi_2) \wedge \neg(\varphi_2 \wedge \neg\varphi_1)$", "$\top$" (resp. "$\bot$") for the true (resp. false) constant. A Σ-literal is either a Σ-atom (a positive literal) or its negation (a negative literal). We call a Σ-formula quantifier-free if it does not contain quantifiers. A quantifier-free formula

Table 1.1: Axioms defining =.

$\forall x.(x = x)$	(reflexivity)
$\forall x, y.(x = y \rightarrow y = x)$	(symmetry)
$\forall x, y, z.(x = y \wedge y = z \rightarrow x = z)$	(transitivity)
$\forall x_1, \ldots, x_n, y_1, \ldots, y_n.(\bigwedge_i x_i = y_i) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$	
$\forall x_1, \ldots, x_n, y_1, \ldots, y_n.(\bigwedge_i x_i = y_i) \rightarrow (P(x_1, \ldots, x_n) \leftrightarrow P(y_1, \ldots, y_n))$	(congruence)

is in conjunctive normal form (CNF) if it is written as a conjunction of disjunctions of literals. A disjunction of literals is called a clause.

We assume the usual first-order notions of interpretation, satisfiability, validity, logical consequence, and theory, as given, e.g., in [End01]. We write $\Gamma \models \varphi$ to denote that the formula φ is a logical consequence of the (possibly infinite) set Γ of formulae. A Σ-theory is a set of first-order sentences with signature Σ. All the theories we consider are first-order theories *with equality*, which means that the equality symbol $=$ is a predefined predicate and it is always interpreted as the identity on the underlying domain. Consequently, $=$ is interpreted as a relation which is reflexive, symmetric, transitive, and it is also a congruence. Therefore, every theory contains the axioms of Table 1.1, for every function symbol f and every predicate symbol P.

A Σ-structure $\mathcal{I}$ is a model of a Σ-theory $\mathcal{T}$ if $\mathcal{I}$ satisfies every sentence in $\mathcal{T}$. A Σ-formula is *satisfiable in* $\mathcal{T}$ (or $\mathcal{T}$-satisfiable) if it is satisfiable in a model of $\mathcal{T}$. A Σ-formula is *valid in* $\mathcal{T}$ (or $\mathcal{T}$-valid) if it is satisfiable in all models of $\mathcal{T}$. We write $\Gamma \models_{\mathcal{T}} \varphi$ to denote $\mathcal{T} \cup \Gamma \models \varphi$. Two Σ-formulae φ and ψ are $\mathcal{T}$-equisatisfiable if and only if φ is $\mathcal{T}$-satisfiable if and only if ψ is $\mathcal{T}$-satisfiable.

A conjunction Γ of $\mathcal{T}$-literals in a theory $\mathcal{T}$ is *convex* if and only if for each disjunction $\bigvee_{i=1}^{n} x_i = y_i$ we have that $\Gamma \models_{\mathcal{T}} \bigvee_{i=1}^{n} x_i = y_i$ if and only if $\Gamma \models_{\mathcal{T}} x_i = y_i$ for some $i \in \{1, ..., n\}$; a theory $\mathcal{T}$ is *convex* if and only if all the conjunctions of literals are convex in $\mathcal{T}$. A theory $\mathcal{T}$ is *stably-infinite*

if and only if for each $\mathcal{T}$-satisfiable formula φ, there exists a model of $\mathcal{T}$ whose domain is infinite and which satisfies φ. Notice that any convex theory whose models are non-trivial (i.e., the domains of the models have all cardinality strictly greater than one) is stably-infinite (see [BDS02]).

We call *Satisfiability Modulo (the) Theory* $\mathcal{T}$, SMT($\mathcal{T}$), the problem of establishing the $\mathcal{T}$-satisfiability of Σ-formulae, for some background theory $\mathcal{T}$. The SMT($\mathcal{T}$) problem is NP-hard, since it subsumes the problem of checking the satisfiability of Boolean formulae.

In this thesis we restrict our attention to *quantifier-free* Σ-formulae on some Σ-theory $\mathcal{T}$.[1] Therefore, unless otherwise specified, when speaking of decidability of the satisfiability problem in some theory $\mathcal{T}$, we in fact mean decidability of the quantifier-free satisfiability problem in $\mathcal{T}$.

1.1.1 Notation

Notationally, we use the Greek letters φ, ψ to represent $\mathcal{T}$-formulae, the capital letters A_i's and B_i's to represent Boolean atoms, and the Greek letters α, β, γ to represent $\mathcal{T}$-atoms in general, the letters l_i's to represent $\mathcal{T}$-literals, the letters μ, η to represent sets of $\mathcal{T}$-literals. If l is a negative $\mathcal{T}$-literal $\neg\beta$, then by "$\neg l$" we conventionally mean β rather than $\neg\neg\beta$.

In what follows, for simplicity and if not specified otherwise, we might omit the prefix "Σ-" when it is clear from the context, and refer simply to terms, atoms, literals, formulae. Moreover, with a little abuse of notation, we might sometimes denote conjunctions of literals $l_1 \wedge \ldots \wedge l_n$ as sets $\{l_1, \ldots, l_n\}$ and vice versa. If η is $\{l_1, \ldots, l_n\}$, we might also write $\neg\eta$ to mean $\neg l_1 \vee \ldots \vee \neg l_n$. Furthermore, following the standard terminology of the SMT (and SAT) community, we shall refer to predicates of arity zero as *Boolean variables*, and to uninterpreted constants as *theory variables*

[1] Notice that in SMT($\mathcal{T}$), the free variables are implicitly existentially quantified, and hence equivalent to Skolem constants.

(or $\mathcal{T}$-variables).

Finally, we define the following functions. The function Atoms(φ) takes a $\mathcal{T}$-formula φ and returns the set of distinct atomic formulae (atoms) occurring in the $\mathcal{T}$-formula φ. The bijective function $\mathcal{T}2\mathcal{B}$ ("Theory-to-Boolean") and its inverse $\mathcal{B}2\mathcal{T} \stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}^{-1}$ ("Boolean-to-Theory") are such that $\mathcal{T}2\mathcal{B}$ maps Boolean atoms into themselves and non-Boolean $\mathcal{T}$-atoms into fresh Boolean atoms — so that two atom instances in φ are mapped into the same Boolean atom if and only if they are syntactically identical — and extend to $\mathcal{T}$-formulae and sets of $\mathcal{T}$-formulae in the obvious way — i.e., $\mathcal{B}2\mathcal{T}(\neg\varphi_1) \stackrel{\text{def}}{=} \neg\mathcal{B}2\mathcal{T}(\varphi_1)$, $\mathcal{B}2\mathcal{T}(\varphi_1 \bowtie \varphi_2) \stackrel{\text{def}}{=} \mathcal{B}2\mathcal{T}(\varphi_1) \bowtie \mathcal{B}2\mathcal{T}(\varphi_2)$ for each Boolean connective $\bowtie$, $\mathcal{B}2\mathcal{T}(\{\varphi_i\}_i) \stackrel{\text{def}}{=} \{\mathcal{B}2\mathcal{T}(\varphi_i)\}_i$. We might sometimes use the superscript ^{p} to denote an application of $\mathcal{T}2\mathcal{B}$: given a $\mathcal{T}$-expression e, we write e^p to denote $\mathcal{T}2\mathcal{B}(e)$, and vice versa. (In the following, we refer to e^p as the *Boolean skeleton* of e.) If $\mathcal{T}2\mathcal{B}(\mu) \models \mathcal{T}2\mathcal{B}(\varphi)$, then we say that μ *propositionally satisfies* φ, written $\mu \models_p \varphi$.

1.2 $\mathcal{T}$-solvers

We call a *theory solver* for $\mathcal{T}$ ($\mathcal{T}$-solver) any procedure establishing whether any given finite conjunction of quantifier-free Σ-literals (or equivalently, any given finite set of Σ-literals) is $\mathcal{T}$-satisfiable ($\mathcal{T}$-consistent) or not.

Besides deciding $\mathcal{T}$-satisfiability, modern $\mathcal{T}$-solvers support several other features which are relevant to SMT($\mathcal{T}$). In what follows, we shall recall the most important ones.

Model Generation. When invoked on a $\mathcal{T}$-satisfiable set of $\mathcal{T}$-literals μ, a model-generating $\mathcal{T}$-solver has the capability of returning a $\mathcal{T}$-model $\mathcal{I}$ which can be used as a witness for the consistency of μ in $\mathcal{T}$ (i.e. $\mathcal{I} \models_{\mathcal{T}} \mu$).

Conflict Set Generation. Given a $\mathcal{T}$-unsatisfiable set of $\mathcal{T}$-literals μ, a

theory conflict set is a $\mathcal{T}$-unsatisfiable subset η of μ. η is a *minimal theory conflict set* if and only if all its strict subsets $\eta' \subset \eta$ are $\mathcal{T}$-satisfiable. As we shall see in §1.3, a crucial factor for the performance of a $\mathcal{T}$-solver in an SMT($\mathcal{T}$) context is its capability of producing small (ideally minimal) theory conflict sets.

Incrementality. Within an SMT($\mathcal{T}$) context (see §1.3), it is often the case that $\mathcal{T}$-solvers are invoked sequentially on *incremental* assignments, in a stack-based manner, like in the following trace (left column first, then right):

$\mathcal{T}$-solver (μ_1)	$\Rightarrow$ sat	Undo μ_4, μ_3, μ_2	
$\mathcal{T}$-solver $(\mu_1 \cup \mu_2)$	$\Rightarrow$ sat	$\mathcal{T}$-solver $(\mu_1 \cup \mu_2')$	$\Rightarrow$ sat
$\mathcal{T}$-solver $(\mu_1 \cup \mu_2 \cup \mu_3)$	$\Rightarrow$ sat	$\mathcal{T}$-solver $(\mu_1 \cup \mu_2' \cup \mu_3')$	$\Rightarrow$ sat
$\mathcal{T}$-solver $(\mu_1 \cup \mu_2 \cup \mu_3 \cup \mu_4)$	$\Rightarrow$ unsat	...	

Thus, a key efficiency issue of $\mathcal{T}$-solvers is that of being *incremental* and *backtrackable*. Incremental means that a $\mathcal{T}$-solver "remembers" its computation status from one call to the other, so that, whenever it is given in input an assignment $\mu_1 \cup \mu_2$ such that μ_1 has just been proved $\mathcal{T}$-satisfiable, it avoids restarting the computation from scratch by restarting the computation from the previous status. Backtrackable means that it is possible to undo steps and return to a previous status on the stack in an efficient manner.

Deduction of Unassigned Literals. Given a set of $\mathcal{T}$-literals μ', and a $\mathcal{T}$-satisfiable subset μ of μ', a $\mathcal{T}$-solver is said to have *deduction* capabilities if, when invoked on μ, it is able to discover one (or more) deductions in the form $\{l_1, \ldots, l_n\} \models_{\mathcal{T}} l$, such that $\{l_1, \ldots, l_n\} \subseteq \mu$ and $l \in \mu' \setminus \mu$. We say that a $\mathcal{T}$-solver is *deduction-complete* if it can perform all possible such deductions, or say that no such deduction can be performed.

1.3 Modern Lazy SMT Solvers

The currently most popular approach for solving the SMT($\mathcal{T}$) problem is the so-called *"lazy"* approach [Seb07, BSST09] (also frequently called "DPLL($\mathcal{T}$)" [NOT06]).

The lazy approach works by combining a propositional SAT solver based on the DPLL algorithm [DLL62] with a $\mathcal{T}$-solver. Essentially, DPLL is used as an enumerator of truth assignments μ_i^p propositionally satisfiying the Boolean skeleton φ^p of the input formula φ, and the $\mathcal{T}$-solver is used for checking the $\mathcal{T}$-satisfiability of each $\mu_i \stackrel{\text{def}}{=} \mathcal{B}2\mathcal{T}(\mu_i^p)$: if the current μ_i is $\mathcal{T}$-satisfiable, then φ is $\mathcal{T}$-satisfiable; otherwise, if none of the μ_i's are $\mathcal{T}$-satisfiable, then φ is $\mathcal{T}$-unsatisfiable.

1.3.1 The Online Lazy SMT Schema

Figure 1.1 represents the schema of a modern lazy SMT solver based on a DPLL engine (see e.g. [ZM02]). It is an abstraction of the algorithm implemented in most state-of-the-art lazy SMT solvers, including BARCELOGIC [BNO+08a], CVC3 [BT07], DPT [GKF08], OPENSMT [BPST09], YICES [DdM06a], Z3 [dMB08c] and MATHSAT.

The input φ and μ are a $\mathcal{T}$-formula and a reference to an (initially empty) set of $\mathcal{T}$-literals respectively. The DPLL solver embedded in $\mathcal{T}$-DPLL reasons on and updates φ^p and μ^p, and $\mathcal{T}$-DPLL maintains some data structure encoding the bijective mapping $\mathcal{T}2\mathcal{B}/\mathcal{B}2\mathcal{T}$ on atoms.

$\mathcal{T}$-preprocess simplifies φ into a simpler formula, and updates μ if it is the case, so that to preserve the $\mathcal{T}$-satisfiability of $\varphi \wedge \mu$. If this process produces some conflict, then $\mathcal{T}$-DPLL returns unsat. $\mathcal{T}$-preprocess may combine most or all the Boolean preprocessing steps available from SAT literature with theory-dependent rewriting steps on the $\mathcal{T}$-literals of φ. This step involves also the conversion to CNF of the

```
SatValue T-DPLL (T-formula φ, reference T-assignment μ)
1.  if T-preprocess (φ, μ) == conflict then
2.      return unsat
3.  end if
4.  φ^p = T2B (φ)
5.  μ^p = T2B (μ)
6.  loop
7.      T-decide-next-branch (φ^p, μ^p)
8.      loop
9.          status = T-deduce (φ^p, μ^p)
10.         if status == sat then
11.             μ = B2T (μ^p)
12.             return sat
13.         else if status == conflict then
14.             blevel = T-analyze-conflict (φ^p, μ^p)
15.             if blevel == 0 then
16.                 return unsat
17.             else
18.                 T-backtrack (blevel, φ^p, μ^p)
19.             end if
20.         else
21.             break
22.         end if
23.     end loop
24. end loop
```

Figure 1.1: An online schema of $\mathcal{T}$-DPLL based on modern DPLL.

input formula, if required.

$\mathcal{T}$-decide-next-branch selects some literal l^p and adds it to μ^p. It plays the same role as the standard literal selection heuristic decide-next-branch in DPLL [ZM02], but it may take into consideration also the semantics in $\mathcal{T}$ of the literals to select. (This operation is called decision, l^p is called decision literal and the number of decision literals in μ after this operation is called the decision level of l^p.)

$\mathcal{T}$-deduce, in its simplest version, behaves similarly to deduce in DPLL [ZM02]: it iteratively deduces Boolean literals l^p which derive propositionally from the current assignment (i.e., s.t. $\varphi^p \wedge \mu^p \models l^p$) and

updates φ^p and μ^p accordingly. (The iterative application of unit-propagation performed by deduce and $\mathcal{T}$-deduce is often called Boolean Constraint Propagation, BCP.) This step is repeated until one of the following facts happens:

(i) μ^p propositionally violates φ^p ($\mu^p \wedge \varphi^p \models \bot$). If so, $\mathcal{T}$-deduce behaves like deduce in DPLL, returning conflict.

(ii) μ^p satisfies φ^p ($\mu^p \models \varphi^p$). If so, $\mathcal{T}$-deduce invokes $\mathcal{T}$-solver on $\mathcal{B}2\mathcal{T}$ (μ^p): if $\mathcal{T}$-solver returns sat, then $\mathcal{T}$-deduce returns sat; otherwise, $\mathcal{T}$-deduce returns conflict.

(iii) no more literals can be deduced. If so, $\mathcal{T}$-deduce returns unknown.

A slightly more elaborated version of $\mathcal{T}$-deduce can invoke $\mathcal{T}$-solver on $\mathcal{B}2\mathcal{T}$ (μ^p) also if μ^p does not yet satisfy φ^p: if $\mathcal{T}$-solver returns unsat, then $\mathcal{T}$-deduce returns conflict. (This enhancement is called *Early Pruning*, EP.)

Moreover, during EP calls, if $\mathcal{T}$-solver is able to perform deductions in the form $\eta \models_{\mathcal{T}} l$ s.t. $\eta \subseteq \mu$ and $l^p \stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}(l)$ is an unassigned literal in φ^p, then $\mathcal{T}$-deduce can append l^p to μ^p and propagate it. (This enhancement is called *$\mathcal{T}$-propagation*.)

$\mathcal{T}$-analyze-conflict is an extension of analyze-conflict of DPLL [ZM02]: if the conflict produced by $\mathcal{T}$-deduce is caused by a Boolean failure (case (i) above), then $\mathcal{T}$-analyze-conflict produces a Boolean conflict set η^p and the corresponding value blevel of the decision level where to backtrack; if instead the conflict is caused by a $\mathcal{T}$-inconsistency revealed by $\mathcal{T}$-solver, then $\mathcal{T}$-analyze-conflict produces the Boolean skeleton $\eta^p \stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}(\eta)$ of the $\mathcal{T}$-conflict set η produced by $\mathcal{T}$-solver. As already mentioned in §1.2, it is important for performance that $\mathcal{T}$-solver generates "good" (short, ideally minimal) $\mathcal{T}$-conflict sets, because they

can make DPLL prune a larger portion of the search space.

$\mathcal{T}$-backtrack behaves analogously to backtrack in DPLL [ZM02]: once the conflict set η^p and blevel have been computed, it adds the clause $\neg\eta^p$ to φ^p, either temporarily or permanently, and backtracks up to blevel. (These features are called $\mathcal{T}$-learning and $\mathcal{T}$-backjumping.)

An important improvement of $\mathcal{T}$-deduce is the following: when $\mathcal{T}$-solver is invoked on EP calls and performs a deduction $\eta \models_{\mathcal{T}} l$ (step (iii) above), then the clause $\mathcal{T}2\mathcal{B}(\neg\eta \vee l)$ (called deduction clause) can be added to φ^p, either temporarily or permanently. The deduction clause will be used for the future Boolean search, with benefits analogous to those of $\mathcal{T}$-learning. To this extent, notice that $\mathcal{T}$-propagation can be seen as a unit-propagation on a deduction clause. (As both $\mathcal{T}$-conflict clauses and deduction clauses are $\mathcal{T}$-valid, they are also called $\mathcal{T}$-lemmas.)

A further enhancement of $\mathcal{T}$-deduce is to use a technique called *layering* [ABC$^+$02, BBC$^+$05, BCF$^+$07], which consists of using a collection of $\mathcal{T}$-solvers $S_1, \ldots, S_N$ organized in a *layered hierarchy* of increasing expressivity and complexity. Each solver S_i is able to decide a theory $\mathcal{T}_i$ which is a subtheory of $\mathcal{T}_{i+1}$, and which is less expensive to handle than $\mathcal{T}_{i+1}$. The solver S_N is the only one that can decide the full theory $\mathcal{T}$. If the solver S_i detects an inconsistency, then there is no need of invoking the more expensive solvers $S_{i+1}, \ldots, S_N$, and unsat can be returned immediately.

Another important improvement of $\mathcal{T}$-analyze-conflict and $\mathcal{T}$-backtrack [NOT06] is that of building from $\neg\eta^p$ also a "mixed Boolean+theory conflict clause", by recursively removing non-decision literals l^p from the clause $\neg\eta^p$ (in this case called conflicting clause) by resolving the latter with the clause C_{l^p} which caused the unit-propagation of l^p (called the *antecedent clause* of l^p); if l^p was propagated by $\mathcal{T}$-propagation, then the deduction clause is used as antecedent clause. This is done until the conflict clause

contains no non-decision literal which has been assigned after the last decision (*last-UIP strategy*) or at most one such non-decision literal (*first-UIP strategy*).[2]

On the whole, $\mathcal{T}$-DPLL differs from the DPLL schema of [ZM02] because it exploits:

- an extended notion of deduction and propagation of literals: not only unit propagation ($\mu^p \wedge \varphi^p \models l^p$), but also $\mathcal{T}$-propagation ($\mathcal{B}2\mathcal{T}(\mu^p) \models_{\mathcal{T}} \mathcal{B}2\mathcal{T}(l^p)$);
- an extended notion of *conflict*: not only *Boolean conflict* ($\mu^p \wedge \varphi^p \models \bot$), but also *theory conflict* ($\mathcal{B}2\mathcal{T}(\mu^p) \models_{\mathcal{T}} \bot$), or even *mixed Boolean+theory conflict* ($\mathcal{B}2\mathcal{T}(\mu^p \wedge \varphi^p) \models_{\mathcal{T}} \bot$).

1.4 Some Relevant Theories in SMT

We give an overview of some of the theories of interest in SMT, supported by most state-of-the-art SMT solvers.

1.4.1 Equality and Uninterpreted Functions

The theory of Equality and Uninterpreted Functions ($\mathcal{EUF}$) [3] is the F.O. theory with equality with no restrictions on Σ. $\mathcal{EUF}$ is stably-infinite and convex. The $\mathcal{EUF}$-satisfiability of sets of quantifier-free literals is decidable and polynomial [Ack54].

An $\mathcal{EUF}$-solver can be implemented on top of data structures and algorithms for computing the *congruence closure* of a set of terms. Intuitively, a congruence-closure based $\mathcal{EUF}$-solver can be described as follows. Given a set of equalities E and a set of disequalities D between terms in a set S, it partitions S into disjoint subsets, called *congruence classes*, such that

[2] These are standard techniques for SAT solvers to build the Boolean conflict clauses [ZMMM01].

[3] Simply called "the theory of equality" by some authors.

two terms t_i and t_j are in the same class if and only if $(t_i = t_j)$ follows from the equality axioms of Table 1.1. The set $E \cup D$ is then $\mathcal{EUF}$-inconsistent if and only if there exists a disequality $\neg(t_i = t_j)$ in D such that t_i and t_j belong to the same congruence class.

Congruence closure can be implemented efficiently on top of the standard Union-Find algorithm (see, e.g., [NO07]), providing important features such as incrementality, efficient backtracking, conflict-set generation and deduction of unassigned equalities and disequalities (see, e.g., [DNS05, NO07]). The algorithm in [NO07] extends $\mathcal{EUF}$ with offset values (that is, it can represent expressions like $(t1 = t2 + k)$, $t1$, t_2 being $\mathcal{EUF}$ terms and k being a constant integer value).

1.4.2 Linear Arithmetic

The theory of Linear Arithmetic ($\mathcal{LA}$) on the rationals ($\mathcal{LA}(\mathbb{Q})$) and on the integers ($\mathcal{LA}(\mathbb{Z})$) is the F.O. theory with equality whose atoms are written in the form $(a_1 \cdot x_1 + ... + a_n \cdot x_n \bowtie a_0)$, s.t. $\bowtie \in \{\leq, <, \neq, =, \geq, >\}$, where the a_i's are (interpreted) constant symbols, each labeling one value in $\mathbb{Q}$ and $\mathbb{Z}$ respectively. The atomic expressions are interpreted according to the standard semantics of linear arithmetic on $\mathbb{Q}$ and $\mathbb{Z}$ respectively. (See, e.g., [MZ03] for a more formal definition of $\mathcal{LA}(\mathbb{Q})$ and $\mathcal{LA}(\mathbb{Z})$. Informally, we consider $\mathcal{LA}(\mathbb{Q})$ as the theory consisting of all formulae with atoms in the form above which are true in the standard model of rational numbers, and similarly for $\mathcal{LA}(\mathbb{Z})$.)

$\mathcal{LA}(\mathbb{Q})$ is stably-infinite and convex. The $\mathcal{LA}(\mathbb{Q})$-satisfiability of sets of quantifier-free literals is decidable and polynomial [Kha79]. The main algorithms used are variants of the well-known Simplex and Fourier-Motzkin algorithms. Efficient incremental and backtrackable algorithms for $\mathcal{LA}(\mathbb{Q})$-solvers have been conceived, which can efficiently perform conflict-set generation and deduction of unassigned literals (see, e.g., [DNS05, RS04,

DdM06a]).

$\mathcal{LA}(\mathbb{Z})$ is stably-infinite and non-convex. The $\mathcal{LA}(\mathbb{Z})$-satisfiability of sets of quantifier-free literals is decidable and NP-complete [Pap81]. A popular algorithm among current SMT solvers for deciding $\mathcal{LA}(\mathbb{Z})$ is to combine a Simplex-based solver for $\mathcal{LA}(\mathbb{Q})$ with some forms of branch-and-bound [Sch86], often combined with the Gomory's cutting planes method [DdM06b]. An alternative is to use the Omega test [Pug91], an extension of the Fourier-Motzkin algorithm for the integers, which has however the disadvantage of requiring huge amounts of memory in general. Recently, a novel algorithm that generalizes branch-and-bound has been proposed [DDA09], showing significant improvements over previous approaches used in SMT.

There are two main relevant sub-theories of $\mathcal{LA}$: the theory of *difference logic* and the *Unit-Two-Variable-Per-Inequality* theory.

1.4.3 Difference logic

The theory of difference logic ($\mathcal{DL}$) on the rationals ($\mathcal{DL}(\mathbb{Q})$) and the integers ($\mathcal{DL}(\mathbb{Z})$) is the sub-theory of $\mathcal{LA}(\mathbb{Q})$ (resp. $\mathcal{LA}(\mathbb{Z})$) whose atoms are written in the form $(0 \bowtie x_2 - x_1 + a)$, such that $\bowtie \in \{\leq, <, \neq, =, \geq, >\}$, and the a is an (interpreted) constant symbol labeling one value in $\mathbb{Q}$ and $\mathbb{Z}$ respectively. [4]

All $\mathcal{DL}$ literals can be rewritten in terms of *positive* difference inequalities ($0 \leq y - x + a$) only (see e.g. [Seb07].) [5]

$\mathcal{DL}(\mathbb{Q})$ is stably-infinite and convex. The $\mathcal{DL}(\mathbb{Q})$-satisfiability problem

[4] Notice that also in $\mathcal{DL}(\mathbb{Q})$ we can assume w.l.o.g. that all constant symbols a occurring in the formula are in $\mathbb{Z}$ because, if this is not so, then we can rewrite the whole formula into an equivalently-satisfiable one by multiplying all constant symbols occurring in the formula by their greatest common denominator.

[5] Notice that in $\mathcal{DL}(\mathbb{Z})$ this process require splitting disequalities into the disjunction of two difference inequalities. In $\mathcal{DL}(\mathbb{Q})$, this is not necessary [Seb07].

of sets of quantifier-free difference inequalities is decidable and polynomial;

The main algorithms encode the $\mathcal{DL}(\mathbb{Q})$-satisfiability of sets of difference inequalities into the problem of finding negative cycles into a weighted oriented graph, called constraint graph. Intuitively, a set S of $\mathcal{DL}(\mathbb{Q})$ atoms induces a graph whose vertexes are the variables of the atoms, and there exists an edge $x \xrightarrow{a} y$ for every $(0 \leq y - x + a) \in S$. S is inconsistent if and only if the induced graph has a cycle of negative weight.

Efficient graph-based incremental algorithms for $\mathcal{DL}(\mathbb{Q})$-solvers have been conceived, which can efficiently perform minimal conflict-set generation and $\mathcal{T}$-deduction of unassigned literals [CM06, NO05].

$\mathcal{DL}(\mathbb{Z})$ is stably-infinite and non-convex. As with $\mathcal{DL}(\mathbb{Q})$, the $\mathcal{DL}(\mathbb{Z})$-satisfiability of sets of quantifier-free difference inequalities is decidable and polynomial; as before, adding equalities does not affect the complexity of the problem. Instead, and due to the non-convexity of $\mathcal{DL}(\mathbb{Z})$, the $\mathcal{DL}(\mathbb{Z})$-satisfiability of sets of quantifier-free difference inequalities, equalities and disequalities is NP-complete [LM05]. Once the problem is rewritten as a set of difference inequalities, the algorithms used for $\mathcal{DL}(\mathbb{Z})$-solvers are the same as for $\mathcal{DL}(\mathbb{Q})$ [CM06, NO05].

1.4.4 Unit-Two-Variable-Per-Inequality

The Unit-Two-Variable-Per-Inequality ($\mathcal{UTVPI}$) theory is a subcase of $\mathcal{LA}$ whose atoms can be written in the form $(0 \leq \pm x_2 \pm x_1 + a)$. Notice that $\mathcal{DL}$ is a sub-theory of $\mathcal{UTVPI}$. $\mathcal{UTVPI}$ is stably-infinite, and it is convex over the rationals ($\mathcal{UTVPI}(\mathbb{Q})$) and non-convex over the integers ($\mathcal{UTVPI}(\mathbb{Z})$). The currently most efficient algorithms for $\mathcal{UTVPI}$ (both over the rationals and over the integers) are based on negative cycle detection on an extended constraint graph [Min01, LM05]. Intuitively, the set of $\mathcal{UTVPI}$ constraints is encoded into a set of $\mathcal{DL}$ constraints by introducing two variables x^+ and x^- for each original variable x, representing x and $-x$

respectively, and encoding each constraint with a pair of $\mathcal{DL}(\mathbb{Z})$ constraint (e.g., $(0 \leq x^+ - y^- + a)$ and $(0 \leq y^+ - x^- + a)$ for $(0 \leq x + y + a)$). In the case of $\mathcal{UTVPI}(\mathbb{Q})$, it is then enough to check for negative-weight cycles in the $\mathcal{DL}(\mathbb{Q})$-constraint graph [Min01]. In the case of $\mathcal{UTVPI}(\mathbb{Z})$, also some particular zero-weight cycles must be detected [LM05].

1.4.5 Arrays

The theory of arrays ($\mathcal{AR}$) aims at modeling the behavior of arrays/memories. The signature consists in the two interpreted function symbols store and select, such that $\mathsf{store}(a, i, e)$ represents (the state of) the array resulting from storing an element e into the location of address i of an array a, and $\mathsf{select}(a, i)$ represents the element contained in the array a at location i. $\mathcal{AR}$ is formally characterized by the following axioms (see [MZ03]):

$$\forall a.\forall i.\forall e.\ (\mathsf{select}(\mathsf{store}(a, i, e), i) = e), \tag{1.1}$$

$$\forall a.\forall i.\forall j.\forall e.\ ((i \neq j) \rightarrow \mathsf{select}(\mathsf{store}(a, i, e), j) = \mathsf{select}(a, j)), \tag{1.2}$$

$$\forall a.\forall b.\ (\forall i.(\mathsf{select}(a, i) = \mathsf{select}(b, i)) \rightarrow (a = b)). \tag{1.3}$$

(1.1) and (1.2), called McCarthy's axioms, characterize the intended meaning of store and select, whilst (1.3), called the extensionality axiom, requires that, if two arrays contain the same values in all locations, than they must be the same array. The theory of arrays is called *extensional* if it includes (1.3), *non-extensional* otherwise.

The $\mathcal{AR}$-satisfiability of sets of quantifier-free literals is decidable and NP-complete [SBDL01]. The most common approach for dealing with $\mathcal{AR}$ is to use a *reduction* to $\mathcal{EUF}$ by means of *axiom instantiation*: a sufficient number of instances of the $\mathcal{AR}$-axioms (1.1)–(1.3) (obtained by replacing the universally-quantified variables with with appropriate ground terms occurring in the input formula) are added to the formula, so that the original formula is $\mathcal{AR}$-unsatisfiable if and only if the augmented formula

is $\mathcal{EUF}$-unsatisfiable [KZ05]. This is typically done *lazily* during search, using a "lemmas-on-demand" approach [dMRS02], in order to minimize the number of axiom instances needed [DNS05, SBDL01, GD07, GKF08, BB09a]. A notable exception to the axiom-instantiation approach is the $\mathcal{AR}$-solver of [BNO+08b]. Finally, the works in [BMS06, GNRZ07] discuss decision procedures for extensions of the $\mathcal{AR}$ theory beyond the select and store operations.

1.4.6 Bit vectors

The theory of fixed-width bit vectors ($\mathcal{BV}$) is a F.O. theory with equality which aims at representing Register Transfer Level (RTL) hardware circuits, so that components such as data paths or arithmetical sub-circuits are considered as entities as a whole, rather than being encoded into purely propositional sub-formulae. $\mathcal{BV}$ can also be used to encode software verification problems (see e.g. [GD07]).

In $\mathcal{BV}$ terms indicate fixed-width bit vectors, and are built from variables (e.g., $\mathbf{x}^{[32]}$ indicates a bit vector x of 32 bits) and constants (e.g., $\mathbf{0}^{[16]}$ denotes a vector of 16 0's) by means of interpreted functions representing standard RTL operators: word concatenation (e.g., $\mathbf{0}^{[16]} :: \mathbf{z}^{[16]}$), sub-word selection (e.g., $(\mathbf{x}^{[32]}[20:5])^{[16]}$), modulo-n sum and multiplication (e.g., $\mathbf{x}^{[32]} +_{32} \mathbf{y}^{[32]}$ and $\mathbf{x}^{[16]} \cdot_{16} \mathbf{y}^{[16]}$), bitwise-operators $\mathbf{and}_n$, $\mathbf{or}_n$, $\mathbf{xor}_n$, $\mathbf{not}_n$ (e.g., $\mathbf{x}^{[16]}\mathbf{and}_{16}\ \mathbf{y}^{[16]}$), left and right shift $<<_n$, $>>_n$ (e.g., $\mathbf{x}^{[32]}<<_4$). Atomic expressions can be built from terms by applying interpreted predicates like $\leq_n$, $<_n$ (e.g., $\mathbf{0}^{[32]} \leq_{32} \mathbf{x}^{[32]}$) and equality.

$\mathcal{BV}$ is non-convex and non-stably infinite. The $\mathcal{BV}$-satisfiability of sets of quantifier-free literals is decidable and NP-complete. Several different approaches for $\mathcal{BV}$-satisfiability have been proposed, e.g. [BDL98, BD02, BBC+06a, BCF+07, GD07, BKO+09]. The currently most efficient algorithms are based on word-level preprocessing followed by encoding the

result into pure SAT ("bit blasting") [MSV07, WFG+07, BB09b, JLS09].

1.5 SMT for Combinations of Theories

In many practical applications of SMT, the theory $\mathcal{T}$ under consideration can be expressed as a *combination* of two (or more) simpler theories $\mathcal{T}_1$ and $\mathcal{T}_2$, SMT($\mathcal{T}_1 \cup \mathcal{T}_2$). For instance, an atom of the form $f(x+4y) = g(2x-y)$, that combines uninterpreted function symbols (from $\mathcal{EUF}$) with arithmetic functions (from $\mathcal{LA}(\mathbb{Z})$, see §1.4), could be used to naturally model in a uniform setting the abstraction of some functional blocks in an arithmetic circuit (see e.g. [BD02, BBC+06a]). In order to deal with SMT($\mathcal{T}_1 \cup \mathcal{T}_2$) formulae, current lazy SMT solvers adopt two different approaches:

- Use some *theory combination mechanism* between the two $\mathcal{T}$-solvers for $\mathcal{T}_1$ and $\mathcal{T}_2$; or
- If one of the two theories is that of equality and uninterpreted functions $\mathcal{EUF}$, an SMT($\mathcal{EUF} \cup \mathcal{T}$) problem can be reduced to an equisatisfiable SMT($\mathcal{T}$) one by applying *Ackermann's reduction* [Ack54].

1.5.1 SMT($\mathcal{T}_1 \cup \mathcal{T}_2$) via Theory Combination

The work on combining $\mathcal{T}$-solvers for distinct theories was pioneered by Nelson and Oppen [NO79, Opp80] and Shostak [Sho84]. [6] In particular, Nelson and Oppen established the theoretical foundations onto which most current combined procedures are still based on (hereafter Nelson-Oppen (N.O.) logical framework). They also proposed a general-purpose procedure for integrating $\mathcal{T}_i$-solvers into one combined $\mathcal{T}$-solver (hereafter

[6] Nowadays there seems to be a general consensus on the fact that Shostak's procedure should not be considered as an independent combination method, rather as a collection of ideas on how to implement Nelson-Oppen's combination method efficiently [BDS02, DNS05].

Nelson-Oppen (N.O.) procedure), based on the deduction and exchange of (disjunctions of) equalities between shared variables (*interface equalities*).

Up to a few years ago, the standard approach to SMT($\mathcal{T}_1 \cup \mathcal{T}_2$) was thus to integrate the SAT solver with one combined $\mathcal{T}_1 \cup \mathcal{T}_2$-solver, obtained from two distinct $\mathcal{T}_i$-solvers by means of the N.O. combination procedure. Variants and improvements of the N.O. procedure were implemented in the CVC/CVCLite [BB04], ICS [dMOR$^+$04], Simplify [DNS05], Verifun [FJOS03], Zapato [BCLZ04] lazy SMT tools.

More recently Bozzano et al. [BBC$^+$06b] proposed Delayed Theory Combination (DTC), a novel combination procedure in which each $\mathcal{T}_i$-solver interacts directly and only with the SAT solver, in such a way that part or all of the (possibly very expensive) reasoning effort on interface equalities is delegated to the SAT solver itself. Variants and improvements of the DTC procedure are currently implemented in the CVC3 [BT07, BNOT06], DPT [KG07], [7] Yices [DdM06c], Z3 [dMB08a] and MathSAT lazy SMT tools; in particular, Yices [DdM06c] and Z3 [dMB08a] introduced many important improvements on the DTC schema (e.g., that of generating interface equalities on-demand, and important "model-based" heuristics to drive the Boolean search on the interface equalities); CVC3 [BT07] combines the main ideas from DTC with that of splitting-on-demand [BNOT06], which pushes even further the idea of delegating to the DPLL engine part of the reasoning effort previously due to the $\mathcal{T}_i$-solvers.

Definitions and Theoretical Background

Consider two theories $\mathcal{T}_1$ and $\mathcal{T}_2$ with equality and whose signatures Σ_1 and Σ_2 are disjoint. A $\Sigma_1 \cup \Sigma_2$-term t is an i-term if and only if either it is a variable or it has the form $f(t_1, \ldots, t_n)$, where f is in Σ_i. Notice that a

[7] Notice that, although [KG07] speak of "Nelson-Oppen with DPLL", their formalism implements and further improves the key ideas of DTC.

variable is both a 1-term and a 2-term. A non-variable subterm s of an i-term t is *alien* if s is a j-term, and all superterms of s in t are i-terms, where $i, j \in \{1, 2\}$ and $i \neq j$. An i-term is i-pure if it does not contain alien subterms. An atom (or a literal) is *i-pure* if it contains only i-pure terms and its predicate symbol is either equality or in Σ_i. A $\mathcal{T}_1 \cup \mathcal{T}_2$-formula φ is said to be pure if every atom occurring in the formula is i-pure for some $i \in \{1, 2\}$. (Intuitively, φ is pure if each atom can can be seen as belonging to one theory $\mathcal{T}_i$ only.) Every non-pure $\mathcal{T}_1 \cup \mathcal{T}_2$ formula φ can be converted into an equisatisfiable pure formula φ' by recursively labeling each alien subterm t with a fresh variable v_t, and by adding the atom $(v_t = t)$. E.g.:

$$(f(x + 3y) = g(2x - y)) \implies$$
$$(f(v_{x+3y}) = g(v_{2x-y})) \wedge (v_{x+3y} = x + 3y) \wedge (v_{2x-y} = 2x - y).$$

This process is called *purification*, and is linear in the size of the input formula. Thus, henceforth we assume w.l.o.g. that all input formulae $\varphi \in \mathcal{T}_1 \cup \mathcal{T}_2$ are pure. [8]

If φ is a pure $\mathcal{T}_1 \cup \mathcal{T}_2$ formula, then v is an *interface variable* for φ if and only if it occurs in both 1-pure and 2-pure atoms of φ. An equality $(v_i = v_j)$ is an *interface equality* for φ if and only if v_i, v_j are interface variables for φ. We assume a unique representation for $(v_i = v_j)$ and $(v_j = v_i)$. (Henceforth, we denote the interface equality $(v_i = v_j)$ by "e_{ij}".) Given a set of literals μ, we say that a $\mathcal{T}$-solver is *e_{ij}-deduction complete* if and only if the $\mathcal{T}$-solver is able to detect all the possible deductions in the form $\mu \models_{\mathcal{T}} e$ (if $\mathcal{T}$ is convex) or in the form $\mu \models_{\mathcal{T}} \bigvee_j e_j$ (if $\mathcal{T}$ is not convex), where e, e_j are interface equalities between variables occurring in μ.

[8] Notice that in fact this assumption is not strictly necessary for the theory combination methods that are described here, thanks to some techiques described in [BDS02]. However, for ease of exposition we shall still assume that purification is performed.

Nelson-Oppen Combination

Consider two stably-infinite theories with equality $\mathcal{T}_1$ and $\mathcal{T}_2$ and disjoint signatures Σ_1 and Σ_2 (often called Nelson-Oppen theories) whose quantifier-free satisfiability problem is decidable, and consider a pure conjunction of $\mathcal{T}_1 \cup \mathcal{T}_2$-literals $\mu \stackrel{\text{def}}{=} \mu_{\mathcal{T}_1} \wedge \mu_{\mathcal{T}_2}$ such that $\mu_{\mathcal{T}_i}$ is i-pure for each i. Nelson and Oppen's key observation [NO79] is that μ is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable if and only if it is possible to find two satisfying interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$ such that $\mathcal{I}_1 \models_{\mathcal{T}_1} \mu_{\mathcal{T}_1}$ and $\mathcal{I}_2 \models_{\mathcal{T}_2} \mu_{\mathcal{T}_2}$ which agree on all equalities on the shared variables.

Overall, Nelson-Oppen results reduce the $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiability problem of a set of pure literals μ to that of finding (the arrangement of) an equivalence relation on the shared variables (μ_e) which is consistent with both pure parts of μ. The condition of having only pure conjunctions as input allows to partition the problem into two independent $\mathcal{T}_i$-satisfiability problems $\mu_{\mathcal{T}_i} \wedge \mu_e$, whose $\mathcal{T}_i$-satisfiability can be checked separately. The condition of having stably-infinite theories is sufficient to guarantee enough values in the domain to allow the satisfiability of every possible set of disequalities one may encounter.

A basic architectural schema of SMT($\mathcal{T}_1 \cup \mathcal{T}_2$) via N.O. is described in Figure 1.2. (Here we provide only a high-level description; the reader may refer, e.g., to [NO79, Sho79, FORS01, BDS02, SR02, DNS05] for more details.) We assume that all $\mathcal{T}_i$'s are N.O. theories and their $\mathcal{T}_i$-solvers are e_{ij}-deduction complete. [9]

We consider first the case in which both theories are convex. The combined $\mathcal{T}_1 \cup \mathcal{T}_2$-solver receives from DPLL a pure set of literals μ, and partitions it into $\mu_{\mathcal{T}_1} \cup \mu_{\mathcal{T}_2}$, s.t. $\mu_{\mathcal{T}_i}$ is i-pure, and feeds each $\mu_{\mathcal{T}_i}$ to the respective

[9] Notice that, theoretically speaking, the condition of e_{ij}-deduction completeness is not strictly needed. However, it is needed *in practice*, in order to be able to implement a deterministic version of the N.O. procedure (see e.g. [BBC+06b, BCF+08] for an extensive discussion about this issue.)

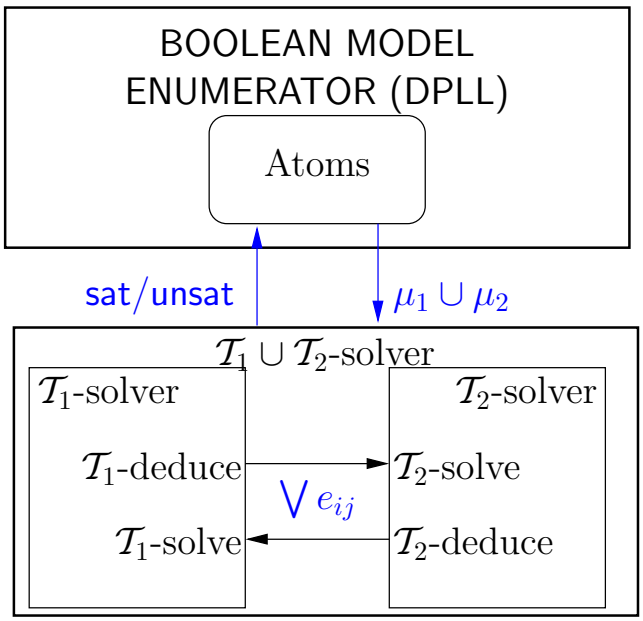


Figure 1.2: A basic architectural schema of SMT($\mathcal{T}_1 \cup \mathcal{T}_2$) via the N.O. procedure.

$\mathcal{T}_i$-solver. Each $\mathcal{T}_i$-solver, in turn:

(i) checks the $\mathcal{T}_i$-satisfiability of $\mu_{\mathcal{T}_i}$,

(ii) deduces all the interface equalities deriving from $\mu_{\mathcal{T}_i}$,

(iii) passes them to the other $\mathcal{T}$-solver, which adds it to his own set of literals.

This process is repeated until either one $\mathcal{T}_i$-solver detects inconsistency ($\mu_1 \cup \mu_2$ is $\mathcal{T}_1 \cup \mathcal{T}_2$-unsatisfiable), or no more e_{ij}-deduction is possible ($\mu_1 \cup \mu_2$ is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable).

In the case in which at least one theory is non-convex, the N.O. procedure becomes more complicated, because the two solvers need to exchange arbitrary disjunctions of interface equalities. As each $\mathcal{T}_i$-solver can handle only conjunctions of literals, the disjunctions must be managed by means of case splitting and of backtrack search. Thus, in order to check the consistency of a set of literals, the combined $\mathcal{T}_1 \cup \mathcal{T}_2$-solver must internally

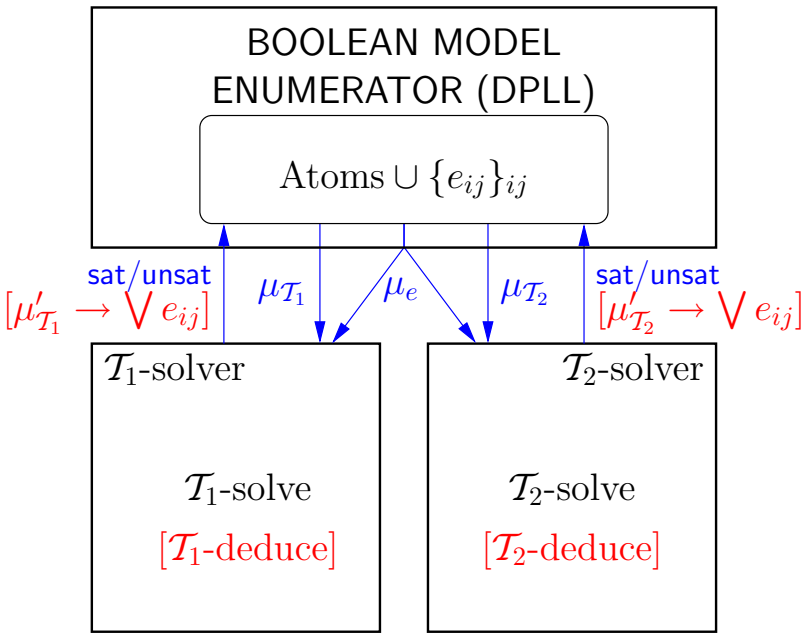


Figure 1.3: A basic architectural schema of SMT($\mathcal{T}_1 \cup \mathcal{T}_2$) via the DTC procedure.

explore a number of branches which depends on how many disjunctions of equalities are exchanged at each step: if the current set of literals is μ, and one of the $\mathcal{T}_i$-solvers sends the disjunction $\bigvee_{k=1}^{n}(e_{ij})_k$ to the other, the latter must further investigate up to n branches to check the consistency of each of the $\mu \cup \{(e_{ij})_k\}$ sets separately.

Delayed Theory Combination

Delayed Theory Combination (DTC) is a more recent general-purpose procedure for tackling the problem of theory combination directly in the context of lazy SMT [BBC+06b, BCF+08]. DTC works by performing Boolean reasoning on interface equalities, possibly combined with $\mathcal{T}$-propagation, with the help of the embedded DPLL solver. As with N.O. procedure, DTC is based on the N.O. logical framework, and thus considers signature-disjoint stably-infinite theories with their respective $\mathcal{T}_i$-solvers, and pure

input formulae (although the consideration on releasing purity — see footnote 8 at page 30 — holds for DTC as well). Importantly, no assumption is made about the e_{ij}-deduction capabilities of the $\mathcal{T}_i$-solvers: for each $\mathcal{T}_i$-solver, every intermediate situation from complete e_{ij}-deduction to no e_{ij}-deduction capabilities is admitted.

A basic architectural schema of DTC is described in Figure 1.3. In DTC, each of the two $\mathcal{T}_i$-solvers interacts directly and only with the Boolean enumerator (DPLL), so that there is no direct exchange of information between the $\mathcal{T}_i$-solvers. The Boolean enumerator is instructed to assign truth values not only to the atoms in Atoms, but also to the interface equalities e_{ij}'s. Consequently, each assignment μ^p enumerated by DPLL is partitioned into three components $\mu^p_{\mathcal{T}_1}$, $\mu^p_{\mathcal{T}_2}$ and μ^p_e, such that each $\mu_{\mathcal{T}_i}$ is the set of i-pure literals and μ_e is the set of interface (dis)equalities in μ, so that each $\mu_{\mathcal{T}_i} \cup \mu_e$ is passed to the respective $\mathcal{T}_i$-solver.

An implementation of DTC [BBC+06b] is based on the online schema of Figure 1.1 in §1.3.1, exploiting early pruning, $\mathcal{T}$-propagation, $\mathcal{T}$-backjumping and $\mathcal{T}$-learning. Each of the two $\mathcal{T}_i$-solvers interacts with the DPLL engine by exchanging literals via the assignment μ in a stack-based manner. The $\mathcal{T}$-DPLL algorithm of Figure 1.1 in §1.3.1 is modified to the following extents:

1. $\mathcal{T}$-DPLL is instructed to assign truth values not only to the atoms in φ, but also to the interface equalities not occurring in φ. $\mathcal{B}2\mathcal{T}$ and $\mathcal{T}2\mathcal{B}$ are modified accordingly.

2. $\mathcal{T}$-decide-next-branch is modified to select also interface equalities e_{ij}'s not occurring in the formula yet.[10]

3. $\mathcal{T}$-deduce is modified to work as follows: instead of feeding the whole

[10]Notice that an interface equality occurs in the formula after a clause containing it is learned, see point 4.

μ to a (combined) $\mathcal{T}$-solver, for each $\mathcal{T}_i$, $\mu_{\mathcal{T}_i} \cup \mu_e$ is fed to the respective $\mathcal{T}_i$-solver. If both return sat, then $\mathcal{T}$-deduce returns sat, otherwise it returns conflict.

4. $\mathcal{T}$-analyze-conflict and $\mathcal{T}$-backtrack are modified so that to use the conflict set returned by one $\mathcal{T}_i$-solver for $\mathcal{T}$-backjumping and $\mathcal{T}$-learning. Importantly, such conflict sets may contain interface (dis)equalities.

5. Early-pruning and $\mathcal{T}$-propagation are performed. If one $\mathcal{T}_i$-solver performs the e_{ij}-deduction $\mu^* \models_{\mathcal{T}_i} \bigvee_{j=1}^{k} e_j$ such that $\mu^* \subseteq \mu_{\mathcal{T}_i} \cup \mu_e$ and each e_j is an interface equality, then the deduction clause $\mathcal{T}2\mathcal{B}(\mu^* \rightarrow \bigvee_{j=1}^{k} e_j)$ is learned.

6. [**If and only if both $\mathcal{T}_i$-solvers are e_{ij}-deduction complete.**] If an assignment μ which propositionally satisfies φ is found $\mathcal{T}_i$-satisfiable for both $\mathcal{T}_i$'s, and neither $\mathcal{T}_i$-solver performs any e_{ij}-deduction from μ, then $\mathcal{T}$-DPLL stops returning sat. [11]

In order to achieve efficiency, other heuristics and strategies have been further suggested in [BBC+06b, BCF+08], and more recently in [BNOT06, DdM06c, dMB08a].

In short, in DTC the embedded DPLL engine not only enumerates truth assignments for the atoms of the input formula, but it also assigns truth values for the interface equalities that the $\mathcal{T}$-solvers are not capable of inferring, and handles the case splits induced by the entailment of disjunctions of interface equalities in non-convex theories. The rationale is to exploit the full power of a modern DPLL engine and to delegate to it part of the heavy reasoning effort on interface equalities previously due to the $\mathcal{T}_i$-solvers. Overall, DTC is simpler to implement than N.O., while at the same time offering several advantages over N.O. in terms of versatility,

[11] This is identical to the $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiability termination condition of N.O. procedure.

efficiency, and restrictions imposed to $\mathcal{T}$-solvers (see [BCF+08] for a comprehensive comparison of N.O. and DTC), and thus it is the combination method of choice for many state-of-the-art SMT solvers, including CVC3 [BT07, BNOT06], DPT [KG07], YICES [DdM06c], Z3 [dMB08c, dMB08a] and MATHSAT.

Model-Based Theory Combination

Model-Based theory combination [dMB08a] is a recent evolution of the DTC idea that was shown to lead to significant improvements in performance. Similarly to DTC, it works by augmenting the Boolean search space with up to all the possible interface equalities. The difference is that the models produced by the single theories $\mathcal{T}_i$ are used to guess the right value for each interface equality, and the DPLL solver is used only if the individual models do not agree on some equality. This approach is based on the observation that in practice *inter-theory* conflicts are much less frequent than *intra-theory* conflicts, and therefore on many cases the models for the individual theories $\mathcal{T}_i$ disagree only on a small subset of all the possible interface equalities. With model-based theory combination, only such mismatches cause extra Boolean search.

More specifically, model-based theory combination works as follows. When given an SMT($\mathcal{T}_1 \cup \mathcal{T}_2$) problem as input, no interface equality is generated, and the $\mathcal{T}_i$-solvers work completely independently. When a complete truth-assignment μ is generated that is consisent in the two individual theories $\mathcal{T}_i$, the models $\mathcal{M}_i$ generated by the two $\mathcal{T}_i$-solvers are checked to discover the implied interface equalities. For each pair of interface variables u and v, only if $\mathcal{M}_i(u) = \mathcal{M}_i(v)$, then the interface equality $u = v$ is generated; moreover, the DPLL engine is instrumented to branch on the true value first. If the number of inter-theory conflicts is small, on most cases this will not cause an inconsistency in the other theory, so no extra

Boolean search will be performed. If this instead leads to a conflict, then DPLL will backjump, flip some literals, and continue searching exactly like in the original DTC. For more details on model-based theory combination and related optimizations, we refer the reader to [dMB08a].

1.5.2 SMT($\mathcal{EUF} \cup \mathcal{T}$) via Ackermann's Reduction

When one of the theories $\mathcal{T}_i$ is $\mathcal{EUF}$, one further approach to the SMT($\mathcal{T}_1 \cup \mathcal{T}_2$) problem is to eliminate uninterpreted function symbols by means of Ackermann's reduction [Ack54] [12] so that to obtain an SMT($\mathcal{T}$) problem with only one theory. The method works by replacing every function application occurring in the input formula φ with a fresh variable and then adding to φ all the needed functional congruence constraints. The new formula φ' obtained is equisatisfiable with φ, and contains no uninterpreted function symbols.

First, each distinct function application $f(t_1, \ldots, t_n)$ is replaced by a fresh variable $v_{f(t_1,\ldots,t_n)}$. Then, for every pair of distinct applications of the same function, $f(t_1, \ldots, t_n)$ and $f(u_1, \ldots, u_n)$, a congruence constraint

$$\bigwedge_{i=1}^{arity(f)} (ack(t_i) = ack(u_i)) \rightarrow (v_{f(t_1,\ldots,t_n)} = v_{f(u_1,\ldots,u_n)}), \tag{1.4}$$

is added, where *ack* is a function that maps each function application $g(w_1, \ldots, w_n)$ into the corresponding variable $v_{g(w_1,\ldots,w_n)}$, each variable into itself and is homomorphic wrt. the interpreted symbols. The atom $(ack(t_i) = ack(u_i))$ is not added if the two sides of the equality are syntactically identical; if so, the corresponding implication in (1.4) is dropped.

[12] often called also *Ackermann's expansion*.

Chapter 2

Details on MathSAT

This chapter is devoted to the engineering aspects of the development of MathSAT, providing details about the implementation of its main components.

Contributions

We provide a detailed description of the main components of MathSAT. In particular, we discuss design choices, heuristics and implementation details that are often omitted from research papers on SMT, but which from our experience can play a significant role in practice for performance. Some of the techniques described are, to the best of our knowledge, peculiar to MathSAT, some of them are instead well-known among developers of SMT solvers. However, most of them have not been previously described in the literature on SMT, and they might not be so obvious to non-experts.

2.1 Overview

Figure 2.1 shows a high-level view of the main components of MathSAT and their interactions.

Preprocessor. Interaction with MathSAT happens either via file, with a variety of input formats supported, or via a rich API. After the

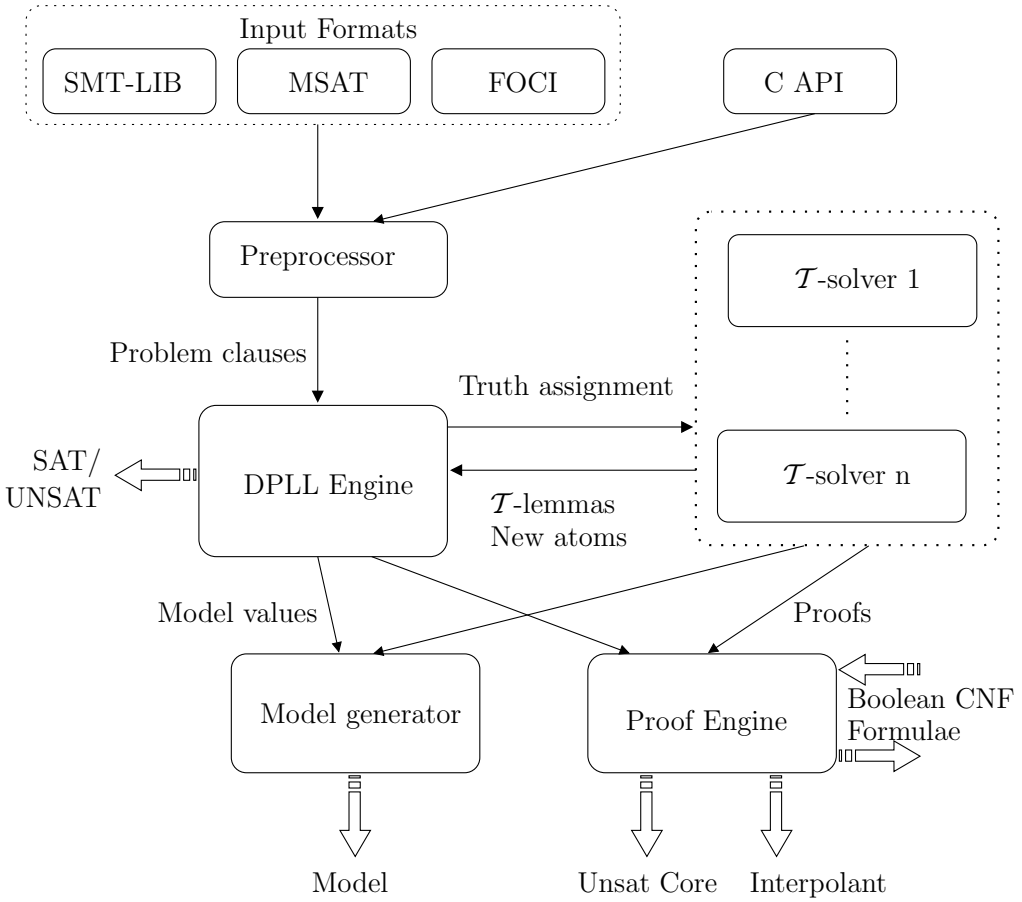


Figure 2.1: Basic schema of MATHSAT architecture.

input formula φ is parsed (or generated through the API), a preprocessing step is performed, in order to simplify the input problem and to convert it to CNF. The main components of the preprocessor are described in §2.2.

DPLL Engine. The core of the solver is the DPLL Engine. It receives as input the CNF conversion of the original problem, and drives the search by enumerating its propositional models and invoking the $\mathcal{T}$-solver(s) to check them for consistency, until either a model is found or all of them are found inconsistent. In §2.3 we describe one particular technique that MATHSAT uses for optimizing the interaction of DPLL and $\mathcal{T}$-solvers.

Theory solvers. In MATHSAT the $\mathcal{T}$-solvers are organized as a *layered hierarchy* of solvers of increasing expressivity and complexity (§1.3.1): if a higher-level solver finds a conflict, then this conflict is used to prune the search at the Boolean level; if it does not, the lower level solvers are activated. These $\mathcal{T}$-solvers implement state-of-the-art procedures for the theories of $\mathcal{EUF}$ [DNS05, NO07], $\mathcal{AR}$ [GKF08], $\mathcal{DL}$ [CM06], $\mathcal{UTVPI}$ [LM05], $\mathcal{LA}(\mathbb{Q})$ [DdM06a], $\mathcal{LA}(\mathbb{Z})$ and their combinations.[1] They are described in §§2.4–2.8.

2.2 The preprocessor

The preprocessing phase of MATHSAT can be divided into three parts.

The first part consists of a series of *satisfiability-preserving* simplification steps of the input formula. By default, MATHSAT performs only basic simplifications, like the encoding of equivalent $\mathcal{T}$-atoms into a unique representation (e.g. the atom $2x+y-x+2 \leq 5$ is converted to $x+y \leq 3$) or the propagation of top-level information (e.g. the formula $x = 5 \wedge f(x) < 3$ is rewritten into $f(5) < 3$). The recent work in [KSJ09] and [Bru09] proposed two more advanced preprocessing techniques, showing that they can have a great positive impact on performance. We are currently investigating such techniques, their limitations, and the possibility of integrating them efficiently into MATHSAT.

The second step is the conversion of the input formula into CNF. This is done using an improved version of the standard algorithm proposed by Tseitin [Tse68]. More sophisticated algorithms have been proposed in the SAT community (e.g. [JS05, MV07]), leading to significant improvements in execution times of SAT solvers. However, when we experimented with

[1] MATHSAT supports also the theory of $\mathcal{BV}$. However, as already mentioned in the Introduction, the support for $\mathcal{BV}$ in MATHSAT was implemented by Anders Franzén, and therefore we shall not describe it in this thesis.

such algorithms, the results we obtained for SMT problems were controversial, showing no clear winner. Therefore, we opted for simplicity and kept the original Tseitin algorithm.

The final preprocessing step is *static learning* [BBC+05, YM06]. Static learning consists in adding to the formula small clauses representing $\mathcal{T}$-valid lemmas (e.g. transitivity constraints) before starting the search. The added lemmas may significantly help to prune the search space in the Boolean level, thus avoiding some calls to the more expensive $\mathcal{T}$-solvers.

2.3 Interaction between the DPLL engine and $\mathcal{T}$-solvers

An efficient interaction between the core DPLL engine and the $\mathcal{T}$-solvers is crucial for the performance of a modern lazy SMT solver. The online schema of Figure 1.1 is however very generic, and it allows for several different strategies and optimizations (see [Seb07] for a survey). Here we describe a specific technique implemented in MATHSAT, which to the best of our knowledge has not been described elsewhere, and discuss its impact on the performance of the system.

2.3.1 Adaptive Early Pruning

Early Pruning (EP, §1.3.1 on page 20) is one of the most important optimizations in the interaction between DPLL and the $\mathcal{T}$-solvers. In particular, EP is effective for two reasons:

(i) it allows to stop exploring branches of the search space which are already $\mathcal{T}$-inconsistent, potentially avoiding the enumeration of an up-to-exponential number of truth assignments; and

(ii) it enables the use of $\mathcal{T}$-propagation (§1.3.1 on page 20), another crucial technique for performance, which further reduces the number of truth assignments to enumerate in DPLL.

The drawback of EP is that $\mathcal{T}$-solvers are called much more frequently, and this can have a substantial cost, especially for some hard theories like $\mathcal{LA}(\mathbb{Z})$. In all the cases in which no $\mathcal{T}$-conflict and no $\mathcal{T}$-deduction are detected, the EP call gives no benefit, and its cost is pure overhead.

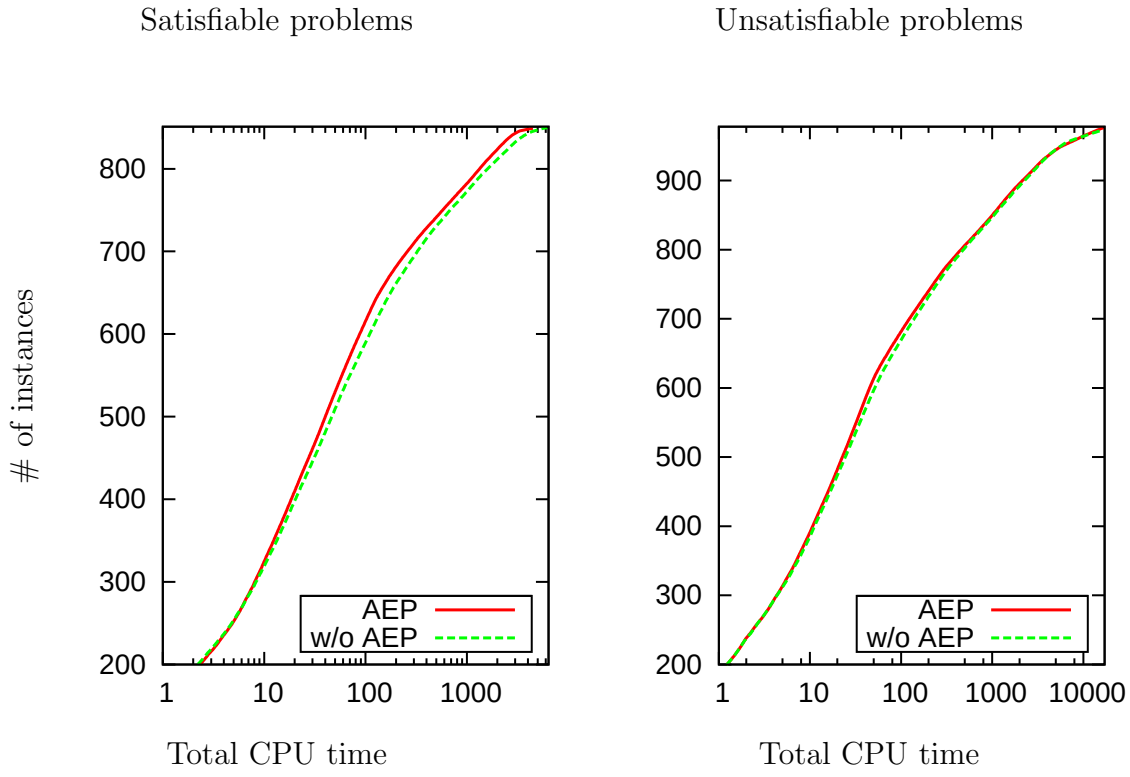
A standard solution for this problem, adopted by several SMT solvers, is to use *incomplete but fast* $\mathcal{T}$-solvers for EP calls, performing the complete but potentially-expensive check only when absolutely necessary (i.e. when a truth assignment which propositionally satisfies the input formula is found). This technique is usually called *Weak (or Approximate) Early Pruning*.

In MathSAT, we have experimented also with a different approach, which we call *Adaptive Early Pruning (AEP)*. The main idea of AEP is that of controlling the frequency of EP calls, by adapting the rate at which $\mathcal{T}$-solvers are invoked according to some measure of the usefulness of EP: the more EP calls contribute to pruning the search by detecting $\mathcal{T}$-conflicts or $\mathcal{T}$-deductions, the more frequently $\mathcal{T}$-solvers are invoked.

The current implementation is rather simple, and it works as follows. We keep a counter `tsolvers_call_interval`, initially set to 1, which controls the frequency of EP calls with respect to the number of decisions performed by DPLL; after every $\mathcal{T}$-decide-next-branch call (see Figure 1.1 on page 19) we increment another counter `num_branches` by one; at the end of $\mathcal{T}$-deduce, if the value of `num_branches` is equal to the value of `tsolvers_call_interval`, we perform an EP call and reset `num_branches`. We then adjust the value of `tsolvers_call_interval`, according to the result of the EP call:

- If the EP call detected a $\mathcal{T}$-conflict, we reset it to 1;
- If the EP call detected some $\mathcal{T}$-deductions, we half its value [2];

[2] making sure it is at least 1, of course.



Satisfiable problems			
# of solved instances	**Total CPU time**		
	AEP	**w/o AEP**	**Ratio**
300	8.00	8.25	0.97
500	39.82	46.18	0.86
700	255.23	320.89	0.79
850	4481.65	5719.96	0.78

Unsatisfiable problems			
# of solved instances	**Total CPU time**		
	AEP	**w/o AEP**	**Ratio**
300	4.27	4.30	0.99
500	22.49	24.21	0.93
700	121.72	137.36	0.88
974	13613.65	16082.67	0.84
978	17127.89	—	—

Figure 2.2: Comparison between MATHSAT with and without AEP, on instances used for SMT-COMP'09. The plots show the accumulated time (on the x axis) to solve a given number of instances (on the y axis), for satisfiable (left) and unsatisfiable (right) problems respectively.

- If the current and the previous EP calls detected neither a $\mathcal{T}$-conflict nor some $\mathcal{T}$-deductions, we double its value;
- Otherwise, we leave it unchanged.

This strategy is rather simple, and it can definitely be improved. Despite this, however, it already gives some interesting performance improvements. In order to demonstrate this, we have performed a comparison between MATHSAT with the AEP strategy and MATHSAT with "standard" EP, on the benchmark instances used in the SMT-COMP'09 SMT solvers competition. The results, which are reported in Figure 2.2, show not only that AEP allows MATHSAT to solve 4 more unsatisfiable instances within the timeout (set to 900 seconds, as in SMT-COMP'09), but also that AEP leads to a reduction of the total execution of up to 20%, both on satisfiable and unsatisfiable instances. We think that this shows the potential of adaptive techniques, making them worth investigating further, by developing more sophisticated heuristics. We also observe that a similar direction of research is being explored by the SAT community, in which adaptive techniques for controlling the frequency of search restarts in DPLL [Bie08a, SI09] have been recently proposed.

2.4 The $\mathcal{EUF}$-solver

Like in all current SMT solvers, the $\mathcal{EUF}$-solver of MATHSAT is based on congruence closure (§1.4.1). In particular, the implementation bears substantial similarities with the one of the SIMPLIFY theorem prover as described in [DNS05], which provides very detailed information about all the data structures used. [3] The main difference is that the $\mathcal{EUF}$-solver of MATHSAT is capable of generating $\mathcal{EUF}$-lemmas (that is, *explanations* for conflicts and implications), a fundamental feature for its use in a lazy SMT

[3] In [DNS05], the $\mathcal{EUF}$-solver is called E-graph.

approach. This is done by adapting in a straightforward way the technique of [NO07] to the data structures used in MATHSAT. [4] With a simple modification, this technique can also be used to generate detailed *proof trees* for $\mathcal{EUF}$-lemmas in terms of applications of the axioms of equality (see Table 1.1 on page 14).

Due to its efficiency — both theoretical (the time complexity of computing the congruence closure of a set of n constraints is $O(n \log n)$ [NO07]) and practical — the $\mathcal{EUF}$-solver is used not only when dealing with purely-equational problems, but also when dealing with other theories $\mathcal{T}$, as a first, incomplete but cheap, procedure in a *layered* approach [ABC+02, BBC+05]: before invoking the $\mathcal{T}$-solver, consistency is checked with the $\mathcal{EUF}$-solver by simply treating all the $\mathcal{T}$-specific functions and predicates as uninterpreted. In this setting, we have found beneficial to augment the power of the $\mathcal{EUF}$-solver in detecting simple arithmetic conflicts, by making it *aware of* the semantics of *numbers* and (in part) of arithmetic relations between them. In particular, the solver knows that two different numeric constants can never belong to the same congruence class, and therefore it can detect conflicts like $(x = 0) \wedge (y = 1) \wedge (x = y)$. Moreover, the solver knows that some arithmetic predicates imply the disequality of their arguments: for example, whenever constraints like $(x < y)$ or $\neg(x \leq y)$ are asserted, the solver knows that x and y can not belong to the same congruence class. Such features are trivial to implement and cause no overhead to the solver, but at the same time they are helpful in improving the effectiveness of layering. Therefore, they are always active in MATHSAT. We remark that such techniques are different from the handling of integer offsets described in [NO07]: the techniques described here are not com-

[4] In particular, in [NO07] it is assumed that terms are *flat* and *curryfied*, whereas in MATHSAT, like in SIMPLIFY, we work with terms represented as nested *cons-cells* [DNS05]. However, this poses no difficulty for the generation of explanations.

plete, but rather they are just heuristics for improving the effectiveness of $\mathcal{EUF}$-layering, whereas the algorithm of [NO07] is a complete decision procedure for $\mathcal{EUF}$ augmented with integer offsets.

From the point of view of the implementation, the $\mathcal{EUF}$-solver of MATHSAT is relatively straightforward, and it follows closely the very detailed description of the SIMPLIFY E-graph provided in [DNS05]. The only detail that deserves a mention is the management of memory. During execution, the $\mathcal{EUF}$-solver allocates a lot of small objects and makes a heavy use of pointers. Therefore, it is worth using custom memory allocation functions that ensure that objects that are frequently accessed together are stored contiguously (or at least close to each other) in the system RAM, in order to make a more effective use of the system cache memory. In our measurements, the use of a specialized memory allocator instead of the default one lead to a significant improvement in the performance of the $\mathcal{EUF}$-solver.

2.5 The $\mathcal{LA}(\mathbb{Q})$-solver

Traditionally, SMT solvers used some kind of incremental simplex algorithm [Van01] as $\mathcal{T}$-solver for the $\mathcal{LA}(\mathbb{Q})$ theory. Recently, Dutertre and de Moura [DdM06a] have proposed a new simplex-based algorithm, specifically designed for integration in a lazy SMT solver. The algorithm is extremely suitable for SMT, and SMT solvers embedding it were shown to significantly outperform (often by orders of magnitude) the ones based on other simplex variants. This algorithm, which was originally implemented within the YICES SMT solver, is also the one implemented in MATHSAT.

Differently from the case of $\mathcal{EUF}$, for which there exist publications that provide sufficient details for implementing an efficient solver (see §2.4 above), the description of the algorithm of Dutertre and de Moura in [DdM06a] is somewhat high-level, and it leaves room for several different

implementation choices that can have a very big impact on performance. [5] Its current implementation in MATHSAT is the result of several months of careful profiling, tuning, and experimentation with different design choices, data structures and heuristics. In this section, we describe such choices in detail and provide experimental evidence of their positive impact on performance.

2.5.1 High-level view of the Dutertre-de Moura algorithm

One of the most important reasons of the efficiency of the Dutertre and de Moura algorithm is its excellent support for incrementality and backtrackability (see §1.2), which allows for adding and removing constraints during search with a very low computational cost. These features are achieved by imposing some restrictions on the form of the constraints it receives as input. In particular, it requires that the variables x_i are partitioned a priori in two sets, hereafter denoted as $\hat{\mathcal{B}}$ ("initially basic" or "dependent") and $\hat{\mathcal{N}}$ ("initially non-basic" or "independent"), and that the algorithm receives as input only two kinds of constraints: [6]

- a set of *equations* eq_i, one for each $x_i \in \hat{\mathcal{B}}$, of the form $\sum_{x_j \in \hat{\mathcal{N}}} \hat{a}_{ij} x_j + \hat{a}_{ii} x_i = 0$ such that all $\hat{a}_{ij}$'s are numerical constants;
- *elementary atoms* of the form $x_j \geq l_j$ or $x_j \leq u_j$ such that $x_j \in \hat{\mathcal{B}} \cup \hat{\mathcal{N}}$ and l_j, u_j are numerical constants. [7]

Moreover, it assumes that the set of equations eq_i does not change during search: the equations are communicated to the $\mathcal{LA}(\mathbb{Q})$-solver before

[5] This was recently observed also in [Mon09].

[6] Notationally, we use the hat symbol $\hat{}$ to denote the initial value of the generic symbol.

[7] In fact, the inequalities in elementary atoms can also be *strict* (i.e. $x_j < l_j$ or $x_j > u_j$). However, the presence of strict inequalities has no impact on the optimizations discussed in this section. Therefore, for ease of exposition, here we shall assume to deal only with problems involving no strict inequality. We shall return to this issue in §4.2.2, in the context of interpolant generation.

starting the Boolean search, and never removed from it. Only elementary atoms can be added and removed during the Boolean search.

As shown in [DdM06a], it is always possible to apply a satisfiability-preserving preprocessing step upfront, before invoking the algorithm, in order to transform every problem into the above form and to ensure that the condition that only elementary atoms need to be added and removed during search is met.

The algorithm is initialized by using the equations eq_i to build a tableau T:

$$\{x_i = \textstyle\sum_{x_j \in \mathcal{N}} a_{ij} x_j \mid x_i \in \mathcal{B}\}, \tag{2.1}$$

where $\mathcal{B}$ ("basic" or "dependent"), $\mathcal{N}$ ("non-basic" or "independent") and a_{ij} are such that initially $\mathcal{B} \equiv \hat{\mathcal{B}}$, $\mathcal{N} \equiv \hat{\mathcal{N}}$ and $a_{ij} \equiv -\hat{a}_{ij}/\hat{a}_{ii}$.

In order to decide the satisfiability of the input problem, the algorithm performs manipulations of the tableau that move variables from $\mathcal{B}$ to $\mathcal{N}$ and vice versa and change the values of the coefficients a_{ij}, always keeping the tableau T in (2.1) equivalent to its initial version.

In particular, the algorithm maintains a mapping $\beta : \mathcal{B} \cup \mathcal{N} \longmapsto \mathbb{Q}$ representing a candidate model which, at every step, satisfies the following invariants:

$$\forall x_j \in \mathcal{N}, \quad l_j \leq \beta(x_j) \leq u_j, \qquad \forall x_i \in \mathcal{B}, \quad \beta(x_i) = \textstyle\sum_{j \in \mathcal{N}} a_{ij} \beta(x_j). \tag{2.2}$$

The algorithm tries to adjust the values of β and the sets $\mathcal{B}$ and $\mathcal{N}$, and hence the coefficients a_{ij} of the tableau, such that $l_i \leq \beta(x_i) \leq u_i$ holds also for all the x_i's in $\mathcal{B}$. Inconsistency is detected when this is not possible without violating any constraint in (2.2).

Like in all variants of the simplex, the central operation in the Dutertre-de Moura algorithm is *pivoting*: given an equation

$$x_i = \textstyle\sum_{x_j \in \mathcal{N} \setminus \{x_k\}} a_{ij} x_j + a_{ik} x_k \tag{2.3}$$

of T such that $a_{ik} \neq 0$, a pivoting operation

(i) replaces (2.3) in T with

$$x_k = \sum_{x_j \in \mathcal{N} \setminus \{x_k\}} \frac{a_{ij}}{-a_{ik}} x_j + \frac{1}{-a_{ik}} x_i \tag{2.4}$$

(ii) replaces all the equations

$$x_h = \sum_{x_j \in \mathcal{N} \setminus \{x_k\}} a_{hj} x_j + a_{hk} x_k$$

of T such that $a_{hk} \neq 0$ with

$$x_h = \sum_{x_j \in \mathcal{N} \setminus \{x_k\}} a_{hj} x_j + a_{hk} \cdot \left(\sum_{x_j \in \mathcal{N} \setminus \{x_k\}} \frac{a_{ij}}{-a_{ik}} x_j + \frac{1}{-a_{ik}} x_i \right) \tag{2.5}$$

(iii) moves x_i from $\mathcal{B}$ to $\mathcal{N}$ and x_k from $\mathcal{N}$ to $\mathcal{B}$.

A pivoting step is performed whenever there exists a variable $x_i \in \mathcal{B}$ such that the current value of $\beta(x_i)$ is not in the range $[l_i, u_i]$ (where l_i and u_i are the currently-active lower and upper bounds for x_i). When this happens, a variable x_j is selected from the i-th row of the tableau, such that it is possible to perform a pivoting of x_i and x_j and to change β in order to make both $\beta(x_i)$ and $\beta(x_j)$ satisfy their bounds [DdM06a].

Pivoting steps are the most expensive operations of the algorithm, and they constitute its main performance bottleneck. Therefore, it is crucial to avoid them whenever possible, and to use data structures and procedures that make them as efficient as possible. As a matter of fact, the reason why the operations of incrementally adding constraints and of backtracking to a previous consistent state are very efficient is exactly that, thanks to the fact that only elementary atoms are involved, they require no pivoting step at all [DdM06a]. [8]

[8] Notice that here, when speaking about adding a constraint, we only refer to the operation of ensuring that the constraint is taken into account in the next consistency checks, and not to the operation of actually checking the consistency of the augmented set of constraints, which does in general require pivoting steps.

2.5.2 Reducing the cost of pivoting operations

Representation of numbers. An essential requirement for $\mathcal{T}$-solvers is that they must be *correct*: a $\mathcal{T}$-solver call can return unsat only if the input set of constraints is inconsistent. [9] An immediate consequence of this is that the arithmetic operations performed by the $\mathcal{LA}(\mathbb{Q})$-solver must be done in exact arithmetic, using infinite-precision rational numbers, in order to ensure the absence of errors due to rounding and/or overflows. However, the use of such numbers causes in general a very significant overhead, which sometimes can be avoided. In particular, for problems in which the coefficients of the variables have small (in absolute value) numerators and denominators, the use of infinite-precision numbers may not be required. In our current implementation, we use a custom library for rational arithmetic, which uses native integers whenever possible, for which arithmetic operations are very fast, and uses the slower infinite-precision numbers (through the GMP library [GMP]) only when overflow is detected. The same technique has been described also by other authors (see e.g. [Mon09] and [DDA09]), and is currently used by several SMT solvers, including Yices (which, as far as we know, was the first SMT solver to use it), BarceLogic, OpenSMT and Z3. In our experiments, we have seen that on many practical problems the use of infinite-precision is not needed at all, and using GMP numbers instead of integers results in a significant degradation of performance, as we will show in §2.5.4.

Representation of the tableau. From the conceptual point of view, the tableau of equations can be seen as a matrix of rational coefficients, in which each row represents an equation. An obvious representation for it would therefore be a two-dimensional array of rational numbers. However,

[9] For complete calls, also the converse must hold, whereas this is not necessary for approximate calls performed during early pruning (see §1.3.1 and §2.3.1).

problems arising from applications in formal verification are typically very sparse: each constraint involves only a small subset of the variables, and therefore most of the entries of the tableau matrix are zero. In such cases, a sparse matrix representation, in which zero entries are not stored explicitly, is much more efficient (both in memory consumption and in execution time) than the array-based one described above. A simple way of implementing it is to represent each equation with a *hash table*, mapping each variable to its coefficient in the equation, and storing only non-zero entries. Another alternative that we have explored is to represent equations using *arrays of pairs* $\langle \text{variable}, \text{coefficient} \rangle$, sorted by variable (and storing only non-zero coefficients). This representation has the disadvantage of requiring logarithmic time for retrieving an arbitrary coefficient a_{ij} of the tableau matrix, whereas this requires (amortized) constant time when using hash tables. However, using arrays of pairs results in a better memory layout with respect to cache usage (since all elements are stored contiguously). Moreover, iterating over arrays is faster than iterating over hash tables, and this makes the implementation of the pivoting steps described by (2.4) and (2.5) faster. In our experiments, we have seen that overall using arrays of pairs gives a performance advantage.

2.5.3 Reducing the number of pivoting steps

As we have seen in §2.5.1, in the Dutertre-de Moura algorithm a pivoting step is performed whenever there exist a variable $x_i \in \mathcal{B}$ and a variable $x_j \in \mathcal{N}$ such that (i) the current value of $\beta(x_i)$ violates one of the bounds for x_i, and (ii) it is possible to perform a pivoting of x_i and x_j and to change β in order to make both $\beta(x_i)$ and $\beta(x_j)$ satisfy their bounds. In general, the choice of x_i and x_j is not unique: first, there might be several basic variables whose bounds are violated by the current β; second, once x_i has been selected, there might be several nonbasic variables which can

become basic. Although in the worst case the number of pivoting steps required for consistency checking is exponential in the number of variables [Van01], it is well-known that in practice the strategy used for selecting the variables to pivot has a very big impact on the performance of simplex-based algorithms [Van01]. In [DdM06a], it is shown that the algorithm terminates if the *Bland's rule* for selecting the variables to pivot is used, which always picks the smallest variable (according to a fixed ordering) among the candidate ones, since this rule is sufficient to guarantee that there are no cycles in the sequence of pivoting steps. However, the Bland's rule typically performs rather poorly, and several alternative strategies have been proposed in order to reduce the number of pivoting steps [Van01]. In MATHSAT, we have adopted the following *greedy* strategy, whose idea is to try to minimize the number of changes required to the tableau T and to β in order to satisfy all the active bounds on the variables:

- when selecting which variable $x_i \in \mathcal{B}$ to pivot, we pick the one for which $\beta(x_i)$ is closest to the violated bound (that is, the one for which the value $l_i - \beta(x_i)$ or $\beta(x_i) - u_i$, depending on which of the two bounds is violated, is the smallest). When there are two or more basic variables at the same distance from the violated bound, we pick the one whose row in T contains fewer nonbasic variables with non-zero coefficient.

- once x_i has been selected, we pick the nonbasic variable x_j (among those which can be pivoted) which occurs (with non-zero coefficient) in the smallest number of rows, in order to minimize the number of updates (2.5) performed during pivoting.

In our experiments, the use of this strategy gave a substantial performance boost compared to the use of the Bland's rule. However, it might lead to cycles in the sequence of pivoting steps, causing the non-termination

of the algorithm in some cases. In order to avoid this, we apply the greedy strategy only until a given threshold value on the number of pivoting steps is reached, and then revert to the Bland's rule. Currently, we use the number of variables in the input problem as threshold.

2.5.4 Experimental evaluation

In order to evaluate the usefulness of the optimizations described above, we have performed several experiments using benchmarks from the QF_LRA division (that is, quantifier-free $\mathcal{LA}(\mathbb{Q})$-instances) of the SMT-LIB [RT06], the library of SMT benchmarks which is used in the annual SMT solvers competition SMT-COMP [SMT]. We have compared the current version of MATHSAT against four other versions, the first three of which were obtained by disabling each of the optimizations individually, whereas the fourth was obtained by disabling all the optimizations. We ran all the experiments on 2.66 Ghz Intel Xeon machines with 6MB of cache, using 2Gb of memory limit and a timeout of 1200 seconds.

The results are reported in Figures 2.3 and 2.4, which show scatter plots of individual comparisons between each configuration with one of the optimizations disabled and the default one, a scatter plot comparing the "naive" configuration with all optimizations disabled with the default one, an "accumulated time" plot showing the total execution time for solving a given number of instances, and a table with statistics about the ratio between the execution time of each tested configuration and the default one. The results clearly show that each of the described optimizations has a significant positive impact on performance, and that their cumulative effect is extremely visible: when all the optimizations are disabled, MATHSAT takes on average 8 times more time to solve problems in the QF_LRA division, failing to solve 21 instances more than the default configuration within the timeout. This is even more impressive if we consider that the

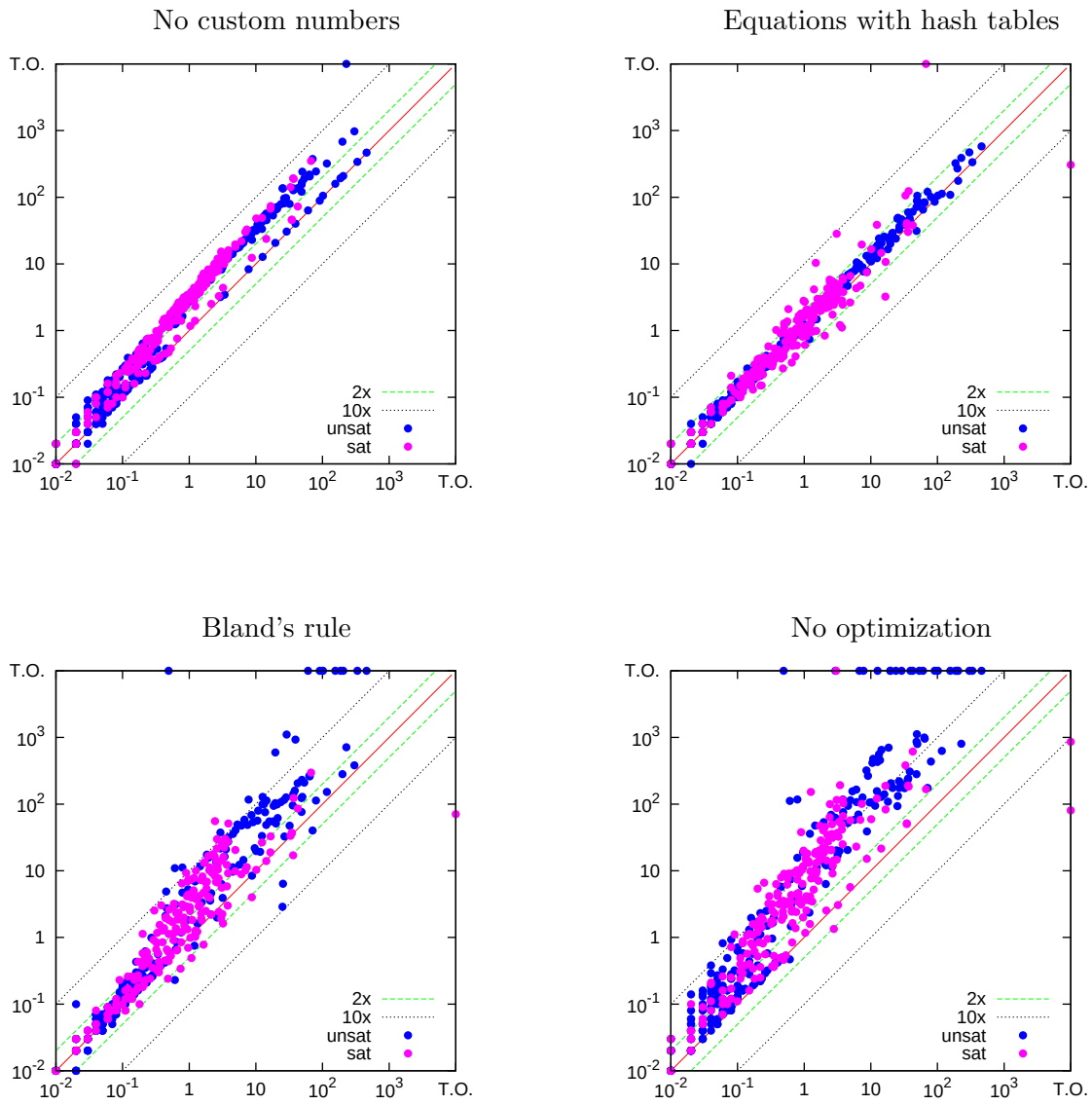
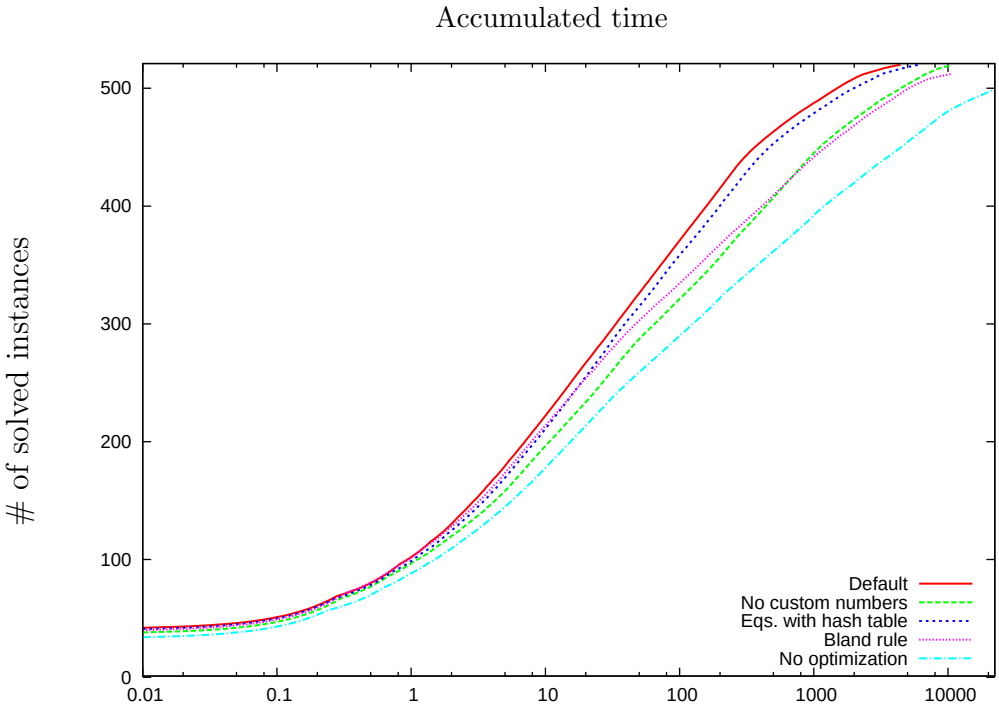


Figure 2.3: Effects of optimizations on $\mathcal{LA}(\mathbb{Q})$-solver performance. The default configuration is always on the x axis, and points on the border of the plot indicate timeouts.



Execution time ratio

Configuration	1^{st} **perc.**	**median**	**mean**	9^{th} **perc.**
No custom numbers	1.00	2.50	2.38	3.71
Eqs. with hash tables	0.95	1.20	1.30	1.75
Bland's rule	0.94	1.42	2.77	6.67
No optimization	1.10	4.48	8.26	17.56

Figure 2.4: Effects of optimizations on $\mathcal{LA}(\mathbb{Q})$-solver performance (continued). The "Accumulated time" plot shows on the y axis the number of instances solved within the timeout, and on the x axis the total execution time. The table shows the most significant percentiles and the mean value of the ratio between the execution time of each configuration and the default one.

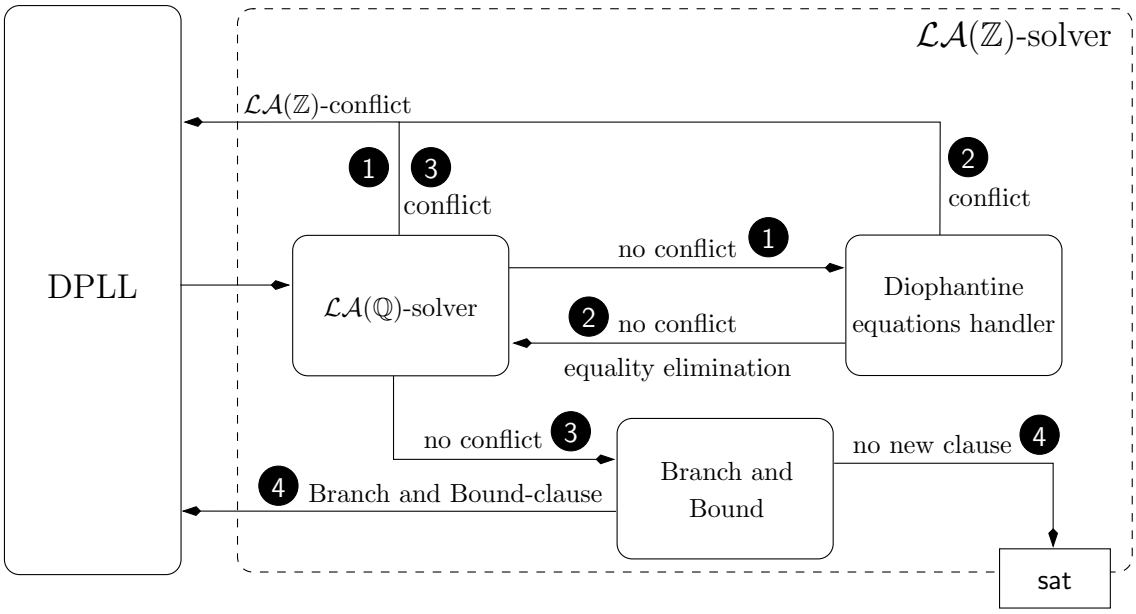


Figure 2.5: Schema of the architecture of the $\mathcal{LA}(\mathbb{Z})$-solver. The numbers in the circles indicate the order of invocation of the various sub-modules.

$\mathcal{LA}(\mathbb{Q})$-solver is only one of the factors that affect the performance of SMT solvers on SMT-LIB instances.

2.6 The $\mathcal{LA}(\mathbb{Z})$-solver

The $\mathcal{LA}(\mathbb{Z})$-solver of MathSAT is one of the most complex parts of the system. Its architecture, outlined in Figure 2.5, is heavily based on layering [ABC+02, BBC+05]: the solver is organized as a hierarchy of sub-modules, with cheaper (but less powerful) ones invoked earlier and more often. In order to check the consistency of a set of $\mathcal{LA}(\mathbb{Z})$-constraints, MathSAT uses the following strategy.

1. First, the real relaxation of the problem is checked, using the $\mathcal{LA}(\mathbb{Q})$-solver described in the previous section. If no conflict is detected, the model returned by the $\mathcal{LA}(\mathbb{Q})$-solver is examined to check whether all variables are assigned to an integer value. If this happens, the $\mathcal{LA}(\mathbb{Q})$-model is also a $\mathcal{LA}(\mathbb{Z})$-model, and the solver can return sat.

2. Otherwise, the specialized module for handling linear Diophantine equations is invoked. This module is similar to the first part of the Omega test described in [Pug91]: it takes all the equations in the input problem, and tries to eliminate them by computing a parametric solution of the system and then substituting each variable in the inequalities with its parametric expression. If the system of equations is infeasible in itself, this module is also able to detect the inconsistency.

3. Otherwise, the inequalities obtained by substituting the variables with their parametric expressions are normalized, tightened and then sent to the $\mathcal{LA}(\mathbb{Q})$-solver, in order to check the $\mathcal{LA}(\mathbb{Q})$-consistency of the new set of constraints.

4. If no conflict is detected, the Branch and Bound module is invoked, which tries to find a $\mathcal{LA}(\mathbb{Z})$-solution via branch and bound [Sch86]. This is done in cooperation with the DPLL engine, using the "splitting on-demand" approach of [BNOT06]: each time the branch and bound algorithm needs to perform a case split, a new clause is generated and sent to the DPLL engine, so that the exploration of the $\mathcal{LA}(\mathbb{Z})$ search space is performed at the Boolean level, thus benefiting for free of all the pruning techniques available in the DPLL engine.

In the rest of the section, we describe in detail the modules for handling Diophantine equations and for performing branch and bound, and discuss their interaction with the $\mathcal{LA}(\mathbb{Q})$-solver and the DPLL engine.

2.6.1 The Diophantine equation handler

The module for handling systems of $\mathcal{LA}(\mathbb{Z})$ equations (commonly called *Diophantine equations*) implements a procedure that closely resembles the equality elimination step of the Omega test [Pug91].

Given a system E of m equations over n variables

$$E \stackrel{\text{def}}{=} \left\{ \sum_{i=1}^{n} a_{ji} x_i + c_j = 0 \right\}_{j=1}^{m} \tag{2.6}$$

it tries to solve it by performing a sequence of variable elimination steps using the procedure described in Algorithm 2.6.

Theorem 2.1. *Algorithm 2.6 always terminates. Moreover, it returns* unsat *if and only if the input system of Diophantine equations is inconsistent.*

Proof. (Sketch) For correctness, we can observe that:

(i) The rewriting of e_h into (2.8) performed in Step 7 is justified by the fact that a_{hk} is the coefficient with the smallest absolute value in e_h.

(ii) At every iteration of the loop 2–7, the initial system E is equisatisfiable with the system $F \cup S$.

(iii) S is always consistent, since all its equations are of the form

$$e_j \stackrel{\text{def}}{=} -x_j + \sum_{i \neq j} a_{ij} x_i + c_j = 0 \tag{2.9}$$

where x_j does not occur in any equation that was added to S after e_j (and therefore it can be easily put in triangular form).

Termination can be established by observing that, after the substitution of x_k with $\sum_{i \neq k} -a^q_{hi} x_i - c^q_h + x_t$ performed in Step 7, the equation e_h of (2.8) becomes

$$a_{hk} x_t + \sum_{i \neq k} a^r_{hi} x_i + c^r_h. \tag{2.10}$$

Since the a^r_{hi}'s are the remainders of the division of the a_{hi}'s by a_{hk}, each $|a^r_{hi}|$ is strictly smaller than the corresponding $|a_{hi}|$. Therefore, after a finite number of applications of Step 7, the equation e_h will contain a

Algorithm 2.6: Solving a system of linear Diophantine equations

1. Let $F = E, S = \emptyset$.
2. If F is empty, the system is consistent; return sat with S as a solution.
3. Rewrite all equations $e_h \stackrel{\text{def}}{=} \sum_i a_{hi}x_i + c_h = 0$ in F such that the GCD g of $a_{h1}, \ldots, a_{hn}, c_h$ is greater than 1 into $e'_h \stackrel{\text{def}}{=} \sum_i \frac{a_{hi}}{g}x_i + \frac{c_h}{g} = 0$.
4. If there exists an equation $e_h \stackrel{\text{def}}{=} \sum_i a_{hi}x_i + c_h = 0$ in F such that the GCD of the a_{hi}'s does not divide c_h, then F is inconsistent (see, e.g., [Pug91]); return unsat.
5. Otherwise, let $e_h \stackrel{\text{def}}{=} \sum_i a_{hi}x_i + c_h = 0$ be an equation, and let a_{hk} be the non-zero coefficient with the smallest absolute value in e_h.
6. If $|a_{hk}| = 1$, then e_h can be rewritten as
$$-x_k + \sum_{i \neq k} -\text{sign}(a_{hk})a_{hi}x_i - \text{sign}(a_{hk})c_h,$$
where $\text{sign}(a_{hk}) \stackrel{\text{def}}{=} \frac{a_{hk}}{|a_{hk}|}$. Then, remove e_h from F, add it to S, and replace x_h with $\sum_{i \neq k} -\text{sign}(a_{hk})a_{hi}x_i - \text{sign}(a_{hk})c_h$ in all the other equations of F.
7. If $|a_{hk}| > 1$, then rewrite e_h as
$$a_{hk}x_k + \sum_{i \neq k}(a_{hk}a^q_{hi} + a^r_{hi})x_i + (a_{hk}c^q_h + c^r_h) \quad \equiv \tag{2.7}$$
$$a_{hk} \cdot (x_k + \sum_{i \neq k} a^q_{hi}x_i + c^q_h) + (\sum_{i \neq k} a^r_{hi}x_i + c^r_h). \tag{2.8}$$
where a^q_{hi} and a^r_{hi} are respectively the quotient and the remainder of the division of a_{hi} by a_{hk} (and similarly for c^q_h and c^r_h). Create a fresh variable x_t, and add to S the equation
$$-x_k + \sum_{i \neq k} -a^q_{hi}x_i - c^q_h + x_t = 0.$$
Then, replace x_k with $\sum_{i \neq k} -a^q_{hi}x_i - c^q_h + x_t$ in all the equations of F.
8. Go to Step 2.

variable whose coefficient has an absolute value of 1, and therefore it will be eliminated from F by an application of Step 6 [Pug91]. □

Example 2.2. Consider the following system of Diophantine equations

$$E \stackrel{\text{def}}{=} \begin{cases} e_1 \stackrel{\text{def}}{=} 3x_1 + 3x_2 + 14x_3 - 7 = 0 \\ e_2 \stackrel{\text{def}}{=} 7x_1 + 12x_2 + 31x_3 - 17 = 0 \end{cases}$$

In order to prove its unsatisfiability, a run of Algorithm 2.6 can proceed as follows:

1. e_1 is processed. Since there are no variables with coefficient 1 or -1, Step 7 is applied. x_1 is selected, e_1 is rewritten as

   $$3(x_1 + x_2 + 4x_3 - 2) + (2x_3 - 1) = 0,$$

   a fresh variable x_4 is created, the equation

   $$-x_1 - x_2 - 4x_3 + 2 + x_4 = 0$$

   is added to S, and x_1 is substituted with $-x_2 - 4x_3 + 2 + x_4$ in all the equations in F, thus obtaining:

   $$S = \left\{ -x_1 - x_2 - 4x_3 + 2 + x_4 = 0 \right.$$

   $$F = \begin{cases} e'_1 \stackrel{\text{def}}{=} 3x_4 + 2x_3 - 1 = 0 \\ e'_2 \stackrel{\text{def}}{=} 5x_2 + 3x_3 + 7x_4 - 3 = 0 \end{cases}$$

2. e'_1 is processed. As before, Step 7 is applied, this time selecting x_3, since it is the variable with the smallest coefficient in absolute value. Then, e'_1 is rewritten as

   $$2(x_3 + x_4) + (x_4 - 1) = 0,$$

   a fresh variable x_5 is created, the equation

   $$-x_3 - x_4 + x_5 = 0$$

is added to S, and x_3 is substituted with $-x_4+x_5$ in all the equations in F, thus obtaining:

$$S = \begin{cases} -x_1 - x_2 - 4x_3 + 2 + x_4 = 0 \\ -x_3 - x_4 + x_5 = 0 \end{cases}$$

$$F = \begin{cases} e_1'' \stackrel{\text{def}}{=} 2x_5 + x_4 - 1 = 0 \\ e_2'' \stackrel{\text{def}}{=} 5x_2 + 4x_4 + 3x_5 - 3 = 0 \end{cases}$$

3. e_1'' is processed. This time, since x_4 has coefficient 1, Step 6 is applied, e_1'' is moved to S and x_4 is substituted with $-2x_5 + 1$ in e_2'', thus obtaining:

$$S = \begin{cases} -x_1 - x_2 - 4x_3 + 2 + x_4 = 0 \\ -x_3 - x_4 + x_5 = 0 \\ -x_4 - 2x_5 + 1 = 0 \end{cases}$$

$$F = \begin{cases} e_2''' \stackrel{\text{def}}{=} 5x_2 - 5x_5 + 1 = 0 \end{cases}$$

4. Since the GCD of the coefficients of the variables in e_2''' does not divide the constant value of e_2''', the equation in inconsistent, so the algorithm returns unsat. ◊

If Algorithm 2.6 returns unsat, the $\mathcal{LA}(\mathbb{Z})$-solver can return unsat. If it returns sat, instead, S can be used to eliminate all the equalities from the problem, using each equation e_j (2.9) as a substitution

$$x_j \mapsto \sum_{i \neq j} a_{ij} x_i + c_j.$$

This elimination is important because it might make possible to *tighten* some of the new inequalities generated. Given an inequality

$$\sum_i a_i x_i \leq c \tag{2.11}$$

such that the GCD g of the a_i's does not divide the constant c, a tightening step [Pug91] consists in rewriting (2.11) into

$$\sum_i \frac{a_i}{g} x_i \leq \lfloor \frac{c}{g} \rfloor. \tag{2.12}$$

Tightening might allow the $\mathcal{LA}(\mathbb{Q})$-solver to detect more conflicts, as shown in the following example.

Example 2.3. Consider the following sets of $\mathcal{LA}(\mathbb{Z})$-constraints:

$$E \stackrel{\text{def}}{=} \begin{cases} 2x_1 - 5x_3 = 0 \\ x_2 - 3x_4 = 0 \end{cases} \qquad I \stackrel{\text{def}}{=} \begin{cases} -2x_1 - x_2 - x_3 \leq -7 \\ 2x_1 + x_2 + x_3 \leq 8 \end{cases}$$

$E \cup I$ is satisfiable over the rationals, but unsatisfiable over the integers. Therefore, the $\mathcal{LA}(\mathbb{Q})$-solver alone can not detect the inconsistency. Thus, E is given to Algorithm 2.6, which returns the following solution:

$$S = \begin{cases} -x_1 + 2x_3 + x_5 = 0 \\ -x_2 + 3x_4 = 0 \\ -x_3 + 2x_5 = 0, \end{cases}$$

where x_5 is a fresh variable. Using S, we can eliminate the equalities by substituting x_1, x_2 and x_3 into the inequalities in I, thus obtaining:

$$I' = \begin{cases} -3x_4 - 12x_5 \leq -7 \\ 3x_4 + 12x_5 \leq 8 \end{cases}$$

On the integers, the two inequalities in I can be *tightened* by dividing the constant by the GCD of the coefficients, and then taking the floor of the result:

$$I'' = \begin{cases} -\frac{3}{3}x_4 - \frac{12}{3}x_5 \leq \lfloor -\frac{7}{3} \rfloor & \text{which becomes } -x_4 - 4x_5 \leq -3 \\ \frac{3}{3}x_4 + \frac{12}{3}x_5 \leq \lfloor \frac{8}{3} \rfloor & \text{which becomes } x_4 + 4x_5 \leq 2 \end{cases}$$

After this, the $\mathcal{LA}(\mathbb{Q})$-solver can immediately detect the inconsistency of I''. $\Diamond$

Generating explanations for conflicts and substitutions

An important capability of the Diophantine equations handler is its ability to produce *explanations* for conflicts, expressed in terms of a subset of the input equations. This is needed not only when an inconsistency is detected by Algorithm 2.6 directly (in order to return to DPLL the corresponding $\mathcal{LA}(\mathbb{Z})$-conflict clause), but also when an inconsistency is detected by the $\mathcal{LA}(\mathbb{Q})$-solver after the elimination of the equalities and the tightening of the inequalities. In this case, in fact, the explanation returned by the $\mathcal{LA}(\mathbb{Q})$-solver can not be used directly to generate a conflict clause to give back to DPLL, since it might contain some inequalities that were generated by the equality elimination and tightening step. When this happens, each of such inequalities must be replaced with the original inequality and the set of equations that were used to obtain it. Therefore, the Diophantine equations handler must be able to identify the set of input equations that were used for generating a substitution in the returned solution S.

In order to describe how explanations are generated and to prove that the procedure is correct, we introduce an abstract transition system whose inference rules mirror the basic steps performed by Algorithm 2.6. We then show how the states and the transitions of such system can be annotated with additional information used to produce explanations. Finally, we give a proof of the correctness of the generated explanations.

The basic steps performed by Algorithm 2.6 can be described as manipulations of a set of equations E according to the following rules:

Scaling of an equation

$$\begin{array}{r} E \cup \{\sum_i a_i x_i + c = 0\} \rightarrow \\ E \cup \{(\sum_i a_i x_i + c = 0), (\sum_i \frac{a_i}{g} x_i + \frac{c}{g} = 0)\} \\ \text{if } g = GCD(a_i, \ldots, a_n, c) \text{ and } g > 1 \end{array} \tag{2.13}$$

Combination of two equations

$$\begin{aligned} E \cup \{(\textstyle\sum_i a_{1i}x_i + c_1 = 0), (\sum_i a_{2i}x_i + c_2 = 0)\} \rightarrow \\ E \cup \{(\textstyle\sum_i a_{1i}x_i + c_1 = 0), (\sum_i a_{2i}x_i + c_2 = 0)\} \cup \\ \{(\textstyle\sum_i (k_1 a_{1i} + k_2 a_{2i})x_i + (k_1 c_1 + k_2 c_2) = 0\}, \\ k_1, k_2 \in \mathbb{Z} \end{aligned} \tag{2.14}$$

Decomposition of an equation

$$\begin{aligned} E \cup \{\textstyle\sum_i a_i x_i + c = 0\} \rightarrow E \cup \{\sum_i a_i x_i + c = 0\} \cup \\ \{(\textstyle\sum_{i \neq k} a_i^q x_i + x_k - x_t + c^q = 0), (a_k x_t + \sum_{i \neq k} a_i^r x_i + c^r = 0)\} \\ \text{if } a_k = \text{argmin}_i\{|a_i| : a_i \neq 0\}, x_t \text{ is fresh}, \\ a_i = a_i^q a_k + a_i^r \text{ for all } i, \text{ and } c = c^q a_k + c^r \end{aligned} \tag{2.15}$$

It is easy to see that Algorithm 2.6 implements a specific strategy of application of the above rules, namely:

- Step 3 corresponds to repeated applications of (2.13);
- Step 6 is an application of (2.14) multiple times; and
- Step 7 corresponds to an application of (2.15) followed by multiple applications of (2.14).

In order to generate explanations, we annotate each state of the above transition system with some additional information. In particular, let X be a set of variables containing all the variables in the initial equations and all the variables introduced by an application of (2.15), let L be a set of variables disjoint from X, and let λ be a mapping from variables in L to linear combinations of variables in X and of integer constants. Moreover, let σ be a partial mapping from variables in X to linear combinations of variables in X and of integer constants. An *annotated state* is a triple $\langle E', \lambda, \sigma \rangle$, where E' is a set of pairs $\langle e, \ell \rangle$ in which e is an equation and ℓ

is a linear combination of variables from L. The initial state of the system is built as follows:

- $E' = \{\langle e_i, l_i\rangle : e_i \in E \text{ and } l_i \in L \text{ is fresh}\}$;
- For all $\langle e_i, l_i\rangle$ in E', set $\lambda(l_i) \mapsto e_i$;
- σ is initially empty.

We define inference rules for annotated states, corresponding to the rules (2.13)–(2.15):

Scaling of an equation

$$\begin{array}{r} \langle E' \cup \{\langle \sum_i a_i x_i + c = 0, \ell\rangle\}, \lambda, \sigma\rangle \rightarrow \\ \langle E' \cup \{\langle \sum_i a_i x_i + c = 0, \ell\rangle, \langle \sum_i \frac{a_i}{g} x_i + \frac{c}{g} = 0, \frac{1}{g}\ell\rangle\}, \lambda, \sigma\rangle \\ \text{if } g = GCD(a_i, \ldots, a_n, c) \text{ and } g > 1 \end{array} \tag{2.16}$$

Combination of two equations

$$\begin{array}{r} \langle E' \cup \{\langle \sum_i a_{1i} x_i + c_1 = 0, \ell_1\rangle, \langle \sum_i a_{2i} x_i + c_2 = 0, \ell_2\rangle\}, \lambda, \sigma\rangle \rightarrow \\ \langle E' \cup \{\langle \sum_i a_{1i} x_i + c_1 = 0, \ell_1\rangle, \langle \sum_i a_{2i} x_i + c_2 = 0, \ell_2\rangle\} \cup \\ \{\langle \sum_i (k_1 a_{1i} + k_2 a_{2i}) x_i + (k_1 c_1 + k_2 c_2) = 0, k_1\ell_1 + k_2\ell_2\rangle\}, \lambda, \sigma\rangle \\ k_1, k_2 \in \mathbb{Z} \end{array} \tag{2.17}$$

Decomposition of an equation

$$\begin{array}{r} \langle E' \cup \{\langle \sum_i a_i x_i + c = 0, \ell\rangle\}, \lambda, \sigma\rangle \rightarrow \langle E' \cup \{\langle \sum_i a_i x_i + c = 0, \ell\rangle\} \cup \\ \{\langle \sum_{i \neq k} a_i^q x_i + x_k - x_t + c^q = 0, 0\ell\rangle, \\ \langle a_k x_t + \sum_{i \neq k} a_i^r x_i + c^r = 0, \ell\rangle\}, \lambda, \sigma'\rangle \\ \text{if } a_k = \text{argmin}_i\{|a_i| : a_i \neq 0\}, x_t \text{ is fresh}, \\ a_i = a_i^q a_k + a_i^r \text{ for all } i, c = c^q a_k + c^r \\ \text{and } \sigma'(x) = \begin{cases} \sum_{i \neq k} a_i^q x_i + x_k + c^q & \text{if } x = x_t \\ \sigma(x) & \text{otherwise} \end{cases} \end{array} \tag{2.18}$$

The purpose of the variables in L and of the mapping λ is to give a name to each of the original equations. We observe in fact that at the beginning each equation is associated to a unique $l_i \in L$ through λ, and that none of the above rules modifies λ. Intuitively, each expression ℓ in a pair $\langle e, \ell \rangle$ of E' encodes the *linear combination of input equations* from which e was generated. When e is inconsistent, therefore, ℓ identifies exactly the subset of input equations responsible for the inconsistency. Analogously, when Algorithm 2.6 returns a solution S, each ℓ_i associated to an equation e_i in S identifies the subset of input equations which were used to generate the substitution encoded by e_i. This argument is formalized by the following theorem.

Theorem 2.4. *Let $\langle E', \lambda, \sigma \rangle$ be an annotated state. Let σ^* be the function that takes a linear combination and recursively replaces each fresh variable x_t introduced by (2.18) with $\sigma(x_t)$, until no more fresh variables are left. Finally, let λ^* be the function that takes a linear combination ℓ of variables from L and replaces each l with $\lambda(l)$. Then, for every element $\langle e, \ell \rangle$ of E', the following holds:*

$$\lambda^*(\ell) = \sigma^*(e) \tag{2.19}$$

Proof. First, observe that (2.19) holds for the initial state of the system, since each element of E' is in the form $\langle e_i, l_i \rangle$ such that $\lambda(l_i) = e_i$. We now show that any application of the rules (2.16)–(2.18) preserves (2.19).

In order to show that this is the case for (2.16) and (2.17), it is enough to observe that, for any linear combinations e_1 and e_2 and coefficients k_1 and k_2, $k_1\sigma^*(e_1) + k_2\sigma^*(e_2) = \sigma^*(k_1 e_1 + k_2 e_2)$, and similarly for λ^*.

As regards (2.18), let $e \stackrel{\text{def}}{=} \langle \sum_i a_i x_i + c = 0, \ell \rangle$ be the element that triggers the application of the rule, and suppose that (2.19) holds for it.

Let

$$\langle e' \stackrel{\text{def}}{=} \sum_{i \neq k} a_i^q x_i + x_k - x_t + c^q = 0, 0\ell \rangle \text{ and} \tag{2.20}$$

$$\langle e'' \stackrel{\text{def}}{=} a_k x_t + \sum_{i \neq k} a_i^r x_i + c^r = 0, \ell \rangle \tag{2.21}$$

be the results of the decomposition, where x_t is fresh. Since (2.18) updates σ by setting $\sigma(x_t) \mapsto \sum_{i \neq k} a_i^q x_i + x_k + c^q$, by the definition of σ^* we have that:

(i)
$$\sigma^*(e') = \sigma^*(\sum_{i \neq k} a_i^q x_i + x_k + c^q - \sigma(x_t)) = \sigma^*(0) = 0,$$
which is clearly equal to $\lambda^*(0\ell) = 0$, and thus (2.19) holds for (2.20); and

(ii)
$$\sigma^*(e'') = \sigma^*(a_k \sigma(x_t) + \sum_{i \neq k} a_i^r x_i + c^r) =$$
$$\sigma^*(a_k(\sum_{i \neq k} a_i^q x_i + x_k + c^q) + \sum_{i \neq k} a_i^r x_i + c^r) = \sigma^*(e),$$
which is equal to $\lambda^*(\ell)$ by hypothesis. Therefore, (2.19) holds also for (2.21). □

Corollary 2.5. *Let $\langle e, \ell \stackrel{def}{=} \sum_i a_i l_i \rangle$ be an element of E', and let E_L be the set of equations $e_i = 0$ such that $\lambda(l_i) = e_i$ for each l_i in ℓ. If e is inconsistent, then E_L is inconsistent.*

Corollary 2.6. *Let S be a solution returned by Algorithm 2.6, let $S' \stackrel{def}{=} \{\langle e_i, \ell_i \stackrel{def}{=} \sum_j a_{ij} l_j \rangle \mid e_i \in S\}$ be its "annotated version", and let E_L be the set of equations $e_j = 0$ such that $\lambda(l_j) = e_j$ for each l_j in $\{\ell_i\}_i$. Let I*

be a set of inequalities, and let I' be the set of inequalities obtained from I after applying the substitutions in S and tightening the results. If I' is inconsistent, then $I \cup E_L$ is inconsistent.

The two corollaries above give us a way of producing explanations with the Diophantine equation handler. Using Corollary 2.5, we can generate an explanation for an inconsistency detected directly by Algorithm 2.6 by taking the conjunction of all the input equations whose labels occur in the linear combination $\sum_i a_i l_i$ associated to the inconsistent equation e. Using Corollary 2.6, instead, we can identify, for each inequality generated by the equality elimination and tightening step, the set of equations used to generate it, by looking at the labels in the linear combinations $\sum_j a_{ij} l_j$ associated to each substitution e_i used. Thanks to this, we can generate an explanation for an inconsistency detected by the $\mathcal{LA}(\mathbb{Q})$-solver by first generating a $\mathcal{LA}(\mathbb{Q})$-explanation containing fresh inequalities generated by the elimination and tightening step, and then by replacing each of such fresh inequalities with the original inequality and the set of equations used to generate it.

2.6.2 The Branch and Bound module

When the equality elimination and tightening step does not lead to an inconsistency, the Branch and Bound module is activated. This module works by scanning the model produced by the $\mathcal{LA}(\mathbb{Q})$-solver in order to find integer variables that were assigned to a rational non-integer constant. If no such variable is found, then the $\mathcal{LA}(\mathbb{Q})$-model is also a $\mathcal{LA}(\mathbb{Z})$-model, and the solver returns sat. Otherwise, let x_k be an integer value to which the $\mathcal{LA}(\mathbb{Q})$-solver has assigned a non-integer value q_k. Then, the Branch and Bound module creates the $\mathcal{LA}(\mathbb{Z})$-lemma $(x_k \leq \lfloor q_k \rfloor) \vee (x_k \geq \lceil q_k \rceil)$, and sends it back to the DPLL engine, which learns it and continues searching. Therefore, the $\mathcal{LA}(\mathbb{Z})$-solver does not always detect conflicts by itself, but

it delegates part of the work to the DPLL engine, following the "splitting on-demand" approach introduced in [BNOT06]. This not only makes the implementation much easier, since there is no need of implementing support for disjunctive reasoning within the $\mathcal{LA}(\mathbb{Z})$-solver, but it also allows to take advantage for free of all the advanced techniques (e.g. conflict-driven backjumping, learning, ...) for search-space pruning implemented in modern DPLL engines.

An important point to highlight is that the branch and bound technique implemented in MATHSAT is not complete, in the sense that it might not terminate (continuing to generate new branch-and-bound-lemmas) if the input problem contains some unbounded variable. Although theoretically it is possible to statically determine bounds for all unbounded variables and thus to make branch and bound complete [Sch86], such theoretical bounds would be so large to have no practical value [DdM06b].

In order to overcome this limitation, we have recently extended the Branch and Bound module with the implementation of the algorithm described in [DDA09], which we refer to for the details. Here, we only mention the fact that this algorithm is based on the computation of proofs of unsatisfiability of systems of Diophantine equations. For this, in [DDA09] the authors use Hermite Normal Forms, whereas in our implementation we can reuse the module for handling Diophantine equations, thanks to its proof-production capability (see Theorem 2.4). Despite the fact that also this method is incomplete unless bounds for all variables are determined a priori, in [DDA09] it was shown to be much more effective than standard branch and bound in practice. However, our implementation is still very recent and basic, and more tuning and experimentation with it is necessary in order to evaluate its benefits.

2.7 Other Theory Solvers

2.7.1 The $\mathcal{AR}$-solver

In order to deal with the theory of arrays, MathSAT adopts an approach based on lazy axiom instantiation (see §1.4.5), implemented on top of the $\mathcal{EUF}$-solver following the algorithm given in [GKF08]. The behaviour of the algorithm can be summarized as follows. Initially, array operations are treated as uninterpreted functions, and a standard congruence closure is used to detect conflicts arising from the violation of equality axioms only. From time to time, an $\mathcal{AR}$-consistency check is performed, in order to detect violations of the $\mathcal{AR}$-axioms (1.1)–(1.3). When a violation is detected, the $\mathcal{AR}$-solver builds a clause corresponding to an instantiation of the violated axiom, and sends it back to the DPLL engine, in order to forbid that specific violation in the future.

The description of the algorithm given in [GKF08] is in terms of a set of inference rules, which leaves a lot of room for exploring several different strategies for deciding *when* and *in which order* to apply them. The strategy that we have adopted – since it was the one which gave the best results in our experiments – consists of deferring the application of the rules until an $\mathcal{EUF}$-consistent complete truth assignment is found, and then to always prefer axioms for (1.1) and (1.2) (read-over-write axioms) over those for (1.3) (extensionality axioms). In other words, we generate extensionality axioms only if no read-over-write axiom is left.

2.7.2 The $\mathcal{DL}$-solver

For Difference Logic, MathSAT implements the algorithm of [CM06], an efficient, incremental, backtrackable, and $\mathcal{T}$-deduction-capable procedure based on detection of negative-weight cycles in a graph-representation of the $\mathcal{DL}$-constraints (see §1.4.3).

As for the case of $\mathcal{LA}(\mathbb{Q})$, in our experiments we have found that the representation of numbers used can have a significant impact on the performance of the solver. Therefore, we adopt the same solution described in §2.5.2: we use native integers by default, and switch to infinite-precision numbers only if needed. In fact, it should be noted that in $\mathcal{DL}$, when using algorithms based on negative-cycle detection in a graph-representation of the constraints, it is possible to determine *statically*, before starting search, whether infinite precision is required or not, by simply summing the absolute values of all the coefficients c of the atoms $(x - y \leq c)$ occurring in the input formula, and checking whether this leads to an overflow.

2.7.3 The $\mathcal{UTVPI}$-solver

Also the solver for $\mathcal{UTVPI}$ constraints is based on the algorithm of [CM06]. For $\mathcal{UTVPI}(\mathbb{Q})$, we simply use the encoding into $\mathcal{DL}(\mathbb{Q})$ given in [Min01] (see §1.4.4 and also §4.4.1) and then use the $\mathcal{DL}$-solver. For $\mathcal{UTVPI}(\mathbb{Z})$, however, this is not enough. Therefore, if the $\mathcal{DL}$-solver returns sat, we use the algorithm of [LM05] for checking consistency over the integers (see also §4.4.2). Since this algorithm is not incremental, we use it only when a complete $\mathcal{UTVPI}(\mathbb{Q})$-consistent truth assignment has been found, whereas during EP calls we only check consistency over the rationals.

2.8 Combination of Theories

MATHSAT supports the combination of $\mathcal{EUF}$ (possibly also with $\mathcal{AR}$– see §2.7.1) with any other theory $\mathcal{T}$, either using the Delayed Theory Combination (DTC) method (§1.5.1) or by applying Ackermann's reduction to eliminate uninterpreted functions and predicates (§1.5.2).[10]

Compared to the original DTC algorithm described in [BBC+06b], the

[10] In this case, this is not possible when $\mathcal{AR}$ is involved.

implementation in MathSAT is significantly improved. In particular, interface equalities are not introduced upfront in the input formula, but only when a truth assignment that is $\mathcal{T}$-consistent in each individual theory is found. Moreover, we use a strategy similar to that described in [BCF+08] for handling case splits on interface equalities, in order to minimize the amount of extra Boolean search induced by the interface equalities. Basically, interface equalities are never selected for case splitting if there is some other unassigned atom, and they are always assigned to false first. However, we do allow $\mathcal{T}$-solvers to $\mathcal{T}$-deduce interface equalities at any time. Finally, we have also implemented a mixed strategy, combining DTC with Ackermann's reduction, based on the Dynamic Ackermannization technique described in [dMB08a]. On several benchmarks of the SMT-LIB, this gives a significant performance improvement.

The support for theory combination could however be still improved. In particular, a promising direction for improvement is the incorporation of ideas from the "Model-Based" combination approach introduced in [dMB08a], which can be seen as an evolution of DTC, and was shown to outperform other combination methods.

Part II

Extended SMT Functionalities

Many important applications of SMT require functionalities that go beyond simply checking the safisfiability of an SMT formula. Examples of such extended functionalities include the ability of producing witnesses for the satisfiability of problems (a model for a satisfiable formula, or a proof of unsatisfiability for an unsatisfiable one), the support for the extraction of unsatisfiable cores and the computation of Craig interpolants, the capability of working incrementally, the enumeration of all the $\mathcal{T}$-consistent truth assignments of a formula (All-SMT), the support for simplifications of formulae and for quantifier elimination.

In particular, in the context of formal verification, such extended functionalities can be very useful for a number of different techniques, among which:

- Models for satisfiable problems can be used for counterexample reconstruction in Bounded Model Checking (BMC) (e.g., [AMP09]), but also for abstraction refinement (e.g., [BH07]).

- Proofs of unsatisfiability and unsatisfiable cores can be used for debugging, for integration with other tools, and for abstraction refinement (e.g., [ABM07, MA03]).

- Craig interpolants can be used for unbounded SAT and SMT based

model checking (e.g., [McM03]), automatic predicate discovery in abstraction refinement (e.g., [HJMM04]), automatic invariant generation (e.g., [McM08]), or simplification of formulae (e.g., [SDPK09]).

- All-SMT capabilities can be used for computing predicate abstractions (e.g. [LNO06, CCF+07]).

MathSAT implements many of the above functionalities: model and proof generation, extraction of unsatisfiable cores, computation of Craig interpolants, All-SMT, and an incremental interface. In particular, MathSAT is (as far as we know) the only modern SMT solver that supports interpolation, and it currently represents the state of the art in interpolant-generation for several important theories. Moreover, it implements a novel procedure for unsatisfiable core extraction, that allows for exploiting for free all the techniques for the extraction of small unsatisfiable cores of propositional formulae, for which several very effective algorithms exist. In this part, we describe in full detail such two distinguishing features of MathSAT. We deal with extraction of unsatisfiable cores in Ch. 3, and with the generation of Craig interpolants in Ch. 4.

Chapter 3

Extraction of Unsatisfiable Cores

Note. *The material presented in this chapter has already been presented in [CGS07] and [CGS09a].*

The concept of *unsatisfiable core* —i.e., an unsatisfiable subset of an unsatisfiable set of clauses— plays a relevant role in SAT-based formal verification, thanks to its many important applications. Examples of such applications include use of SAT instead of BDDs for unbounded symbolic model checking [McM02], automatic predicate discovery in abstraction refinement frameworks [MA03, WKG07], decision procedures [BKO+09], under-approximation and refinement in the context of bounded model checking of multi-threaded systems [GLST05], debugging of design errors in circuits [SFBD08]. For this reason, the problem of finding small unsat cores in SAT has been addressed by many authors in the recent years [LMS04, MLA+05, ZM03, OMA+04, Hua05, DHN06, Bie08b, GKS08, ZLS06, vMW08, ANORC08].

Surprisingly however, the problem of finding unsatisfiable cores in SMT has received virtually no attention in the literature. Although some SMT tools do compute unsat cores, this is done either as a byproduct of the more general task of producing proofs, or by modifying the embedded DPLL solver so that to apply basic propositional techniques to produce an unsat

core. In particular, we are not aware of any work aiming at producing *small* unsat cores in SMT.

Contributions

We address the problem of computing small unsatisfiable cores in SMT, by presenting a novel, SMT-specific approach to unsat core computation, which we call the *Lemma-Lifting approach*. The main idea is to combine an SMT solver with an external propositional core extractor. The SMT solver stores and returns the theory lemmas it had to prove in order to refute the input formula; the external core extractor is then called on the Boolean skeleton of the original SMT problem and of the theory lemmas. The resulting Boolean unsatisfiable core is cleaned from (the Boolean skeleton of) all theory lemmas, and it is refined back into a subset of the original clauses. The result is an unsatisfiable core of the original SMT problem.

We evaluate our approach by an extensive empirical test on SMT-LIB benchmarks, in terms of both effectiveness (reduction in size of the cores) and efficiency (execution time). The results confirm the validity and versatility of this approach.

As a byproduct, we also produce an extensive and insightful evaluation of the main Boolean unsat-core-generation tools currently available.

3.1 State of The Art

3.1.1 Definitions

Without loss of generality, in the following we consider only formulae in CNF. Given an unsatisfiable CNF formula φ, we say that an unsatisfiable CNF formula ψ is an *unsatisfiable core* (UC) of φ if and only if $\varphi = \psi \wedge \psi'$ for some (possibly empty) CNF formula ψ'. Intuitively, ψ is a subset of the clauses in φ causing the unsatisfiability of φ. An unsatisfiable core ψ is

minimal if and only if the formula obtained by removing any of the clauses of ψ is satisfiable. A *minimum* unsat core is a minimal unsat core with the smallest possible cardinality.

The concept of unsatisfiable core is strictly related to that of *proof of unsatisfiability*.

Definition 3.1 (Resolution proof)**.** *Given a set of clauses* $S \stackrel{def}{=} \{C_1, \ldots, C_n\}$ *and a clause* C*, we call a* resolution proof *of the deduction* $\bigwedge_i C_i \models_{\mathcal{T}} C$ *a DAG* $\mathcal{P}$ *such that:*

1. C *is the root of* $\mathcal{P}$*;*
2. *the leaves of* $\mathcal{P}$ *are either elements of* S *or* $\mathcal{T}$*-lemmas;*
3. *each non-leaf node* C' *has two premises* C_{p_1} *and* C_{p_2} *such that* $C_{p_1} \stackrel{def}{=} p \vee \phi_1$*,* $C_{p_2} \stackrel{def}{=} \neg p \vee \phi_2$*, and* $C' \stackrel{def}{=} \phi_1 \vee \phi_2$*. The atom* p *is called the* pivot *of* C_{p_1} *and* C_{p_2}*.*

If C *is the empty clause (denoted with* $\bot$*), then* $\mathcal{P}$ *is a* resolution proof of ($\mathcal{T}$-)unsatisfiability[1] *for* $\bigwedge_i C_i$*.*

3.1.2 Techniques for unsatisfiable-core extraction in SAT

In the last few years, several algorithms for computing small, minimal or minimum unsatisfiable cores of propositional formulae have been proposed. Several techniques work by extracting unsatisfiable cores from resolution refutations generated by a DPLL-based solver (see [ANORC08] for an in-depth discussion and comparison of such approaches). The production of a resolution proof with DPLL can be achieved easily, with very little implementation effort, by exploiting the information and the data structures maintained by DPLL for performing conflict analysis and conflict-driven

[1] often called also *resolution refutation*.

backjumping (such data structures are commonly referred to as *implication graph* [ZM02]; we refer to [vG07] for a concise and clear description of proof-generation with DPLL). In [ZM03], the computed unsat core is simply the collection of all the original clauses that the DPLL solver used to derive the empty clause by resolution. The returned core is not minimal in general, but it can be reduced by iterating the algorithm until a fixpoint, using as input of each iteration the core computed at the previous one. The algorithm of [GKS08], instead, manipulates the resolution proof so that to shrink the size of the core, using also a fixpoint iteration as in [ZM03] to further enhance the quality of the results. In [OMA+04], an algorithm to compute *minimal* unsat cores is presented. The technique is based on modifications of a standard DPLL engine, and works by adding some extra variables (selectors) to the original clauses, and then performing a branch-and-bound algorithm on the modified formula. The procedure presented in [Hua05] extracts minimal cores using BDD manipulation techniques, removing one clause at a time until the remaining core is minimal. The construction of a minimal core in [DHN06] also uses resolution proofs, and it works by iteratively removing from the proof one input clause at a time, until it is no longer possible to prove inconsistency. When a clause is removed, the resolution proof is modified to prevent future use of that clause.

As far as the the computation of *minimum* unsatisfiable cores is concerned, the algorithm of [LMS04] searches all the unsat cores of the input problem; this is done by introducing selector variables for the original clauses, and by increasing the search space of the DPLL solver to include also such variables; then, (one of) the unsatisfiable subformulae with the smallest number of selectors assigned to true is returned. The approach described in [MLA+05] instead is based on a branch-and-bound algorithm that exploits the relation between maximal satisfiability and minimum un-

satisfiability. The same relation is used also by the procedure in [ZLS06], which is instead based on a genetic algorithm.

3.1.3 Techniques for unsatisfiable-core extraction in SMT

To the best of our knowledge, there is no literature on the computation of unsatisfiable cores in SMT. However, at least four SMT solvers (i.e. CVC3 [BT07], YICES [DdM06a], Z3 [dMB08c] and MATHSAT) support unsat core generation [2]. In the following, we describe the underlying approaches, that generalize techniques for propositional UC extraction. We preliminarily remark that none of these solvers aims at producing minimal or minimum unsat cores, nor does anything to reduce their size.

Proof-based UC extraction

CVC3 and MATHSAT can run in proof-producing mode, and compute unsatisfiable cores as a byproduct of the generation of proofs. Similarly to the approach in [ZM03], the idea is to analyze the proof of unsatisfiability backwards, and to return an unsatisfiable core that is a collection of the assumptions (i.e. the clauses of the original problem) that are used in the proof to derive contradiction.

Example 3.2. In order to show how the described approaches work, consider this small unsatisfiable SMT($\mathcal{T}$) formula, where $\mathcal{T}$ is $\mathcal{LA}(\mathbb{Z})$:

$$(x = 0 \vee \neg(x = 1) \vee A_1) \wedge (x = 0 \vee x = 1 \vee A_2) \wedge (\neg(x = 0) \vee x = 1 \vee A_2) \wedge$$
$$(\neg A_2 \vee y = 1) \wedge (\neg A_1 \vee x + y > 3) \wedge (y < 0) \wedge (A_2 \vee x - y = 4) \wedge$$
$$(y = 2 \vee \neg A_1) \wedge (x \geq 0), \quad (3.1)$$

where x and y are integer variables and A_1 and A_2 are Booleans.

[2] The information reported here on the computation of unsat cores in CVC3, YICES and Z3 comes from private communications from the authors and from the user manual of CVC3.

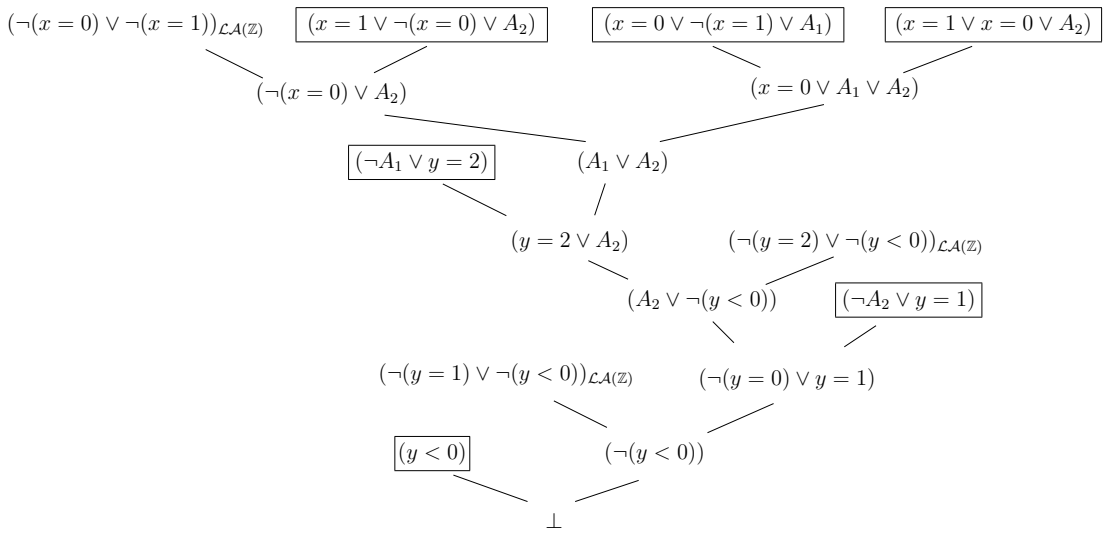


Figure 3.1: Resolution proof for the SMT formula (3.1) found by MathSAT. Boxed clauses correspond to the unsatisfiable core.

In the proof-based approach, a resolution proof of unsatisfiability is built during the search. E.g., Figure 3.1 shows the proof tree found by MathSAT. The leaves of the tree are either original clauses (boxed in the Figure) or $\mathcal{LA}(\mathbb{Z})$-lemmas (denoted with the $\mathcal{LA}(\mathbb{Z})$ suffix). The unsatisfiable core is built by collecting all the original clauses appearing as leaves in the proof. In this case, this is:

$$\{(x = 0 \vee \neg(x = 1) \vee A_1), (x = 0 \vee x = 1 \vee A_2), (\neg(x = 0) \vee x = 1 \vee A_2), (\neg A_2 \vee y = 1), (y < 0), (y = 2 \vee \neg A_1)\}. \quad (3.2)$$

In this case, the unsat core is minimal. $\Diamond$

Assumption-based UC extraction

The approach used by Yices [DdM06a] and Z3 [dMB08c] is an adaptation of the method from [LMS04]: for each clause C_i in the problem, a new Boolean "selector" variable S_i is created; then, each C_i is replaced by $(S_i \rightarrow C_i)$; finally, before starting the search each S_i is forced to true.

In this way, when a conflict at decision level zero is found by the DPLL solver, the conflict clause contains only selector variables, and the unsat core returned is the union of the clauses whose selectors appear in such conflict clause.

Example 3.3. Consider again the formula (3.1) of Example 3.2. In the assumption-based approach, each of the 9 input clauses is augmented with an extra variable S_i, which is asserted to true at the beginning of the search. The formula therefore becomes:

$$\begin{gathered}\bigwedge_i S_i \wedge \\ (S_1 \rightarrow (x = 0 \vee \neg(x = 1) \vee A_1)) \wedge (S_2 \rightarrow (x = 0 \vee x = 1 \vee A_2)) \wedge \\ (S_3 \rightarrow (\neg(x = 0) \vee x = 1 \vee A_2)) \wedge (S_4 \rightarrow (\neg A_2 \vee y = 1)) \wedge \\ (S_5 \rightarrow (\neg A_1 \vee x + y > 3)) \wedge (S_6 \rightarrow y < 0) \wedge \\ (S_7 \rightarrow (A_2 \vee x - y = 4)) \wedge (S_8 \rightarrow (y = 2 \vee \neg A_1)) \wedge (S_9 \rightarrow x \geq 0)\end{gathered} \tag{3.3}$$

The final conflict clause generated by conflict analysis [ZMMM01] is: [3]

$$\neg S_1 \vee \neg S_2 \vee \neg S_3 \vee \neg S_4 \vee \neg S_6 \vee \neg S_7 \vee \neg S_8, \tag{3.4}$$

corresponding to the following unsat core:

$$\begin{aligned}\{(x = 0 \vee \neg(x = 1) \vee A_1), (x = 0 \vee x = 1 \vee A_2), (\neg(x = 0) \vee x = 1 \vee A_2), \\ (\neg A_2 \vee y = 1), (y < 0), (A_2 \vee x - y = 4), (y = 2 \vee \neg A_1)\}.\end{aligned} \tag{3.5}$$

Notice that this is not minimal, because of the presence of the redundant clause $(A_2 \vee x - y = 4)$, corresponding to $\neg S_7$ in the final conflict clause (3.4). ◊

Remark 3.4. *The idea behind the two techniques just illustrated is substantially the same. Both exploit the implication graph built by DPLL during*

[3] using YICES.

conflict analysis to detect the subset of the input clauses that were used to decide unsatisfiability. The main difference is that in the proof-based approach this is done by explicitly constructing the proof tree, while in the activation-based one this can be done "implicitly" by "labeling" each of the original clauses. However, this difference has no impact on the final result. In particular, given the same search strategy*, the two techniques will return the* same unsat core*. The fact that the unsatisfiable cores of Examples 3.2 and 3.3 are different is therefore just a consequence of different Boolean search steps performed by* MathSAT *and* Yices *for that particular formula. For a deeper comparison between these two approaches (and some variants of them), we refer the reader to [ANORC08].*

3.2 A novel approach: Lemma-Lifting

We present a novel approach, called the *Lemma-Lifting approach*, in which the unsatisfiable core is computed *a posteriori* w.r.t. the execution of the SMT solver, and only if the formula has been found $\mathcal{T}$-unsatisfiable. This is done by means of an external (and possibly optimized) propositional unsat-core extractor.

3.2.1 The main ideas

In the following, we assume that a lazy SMT($\mathcal{T}$) procedure has been run over a $\mathcal{T}$-unsatisfiable set of SMT($\mathcal{T}$) clauses $\varphi \stackrel{\text{def}}{=} \{C_1, \ldots, C_n\}$, and that $D_1, \ldots, D_k$ denote all the $\mathcal{T}$-lemmas, both theory-conflict and theory-deduction clauses, which have been returned by the $\mathcal{T}$-solver during the run. In case of mixed Boolean+theory-conflict clauses [NOT06] (see §1.3.1 on page 21), the $\mathcal{T}$-lemmas are those which have been used to compute the mixed Boolean+theory-conflict clause, including the initial theory-conflict clause and the theory-deduction clauses corresponding to the theory-propagation

steps performed. Under the above assumptions, two simple facts hold.

(i) Since the $\mathcal{T}$-lemmas D_i are valid in $\mathcal{T}$, they do not affect the $\mathcal{T}$-satisfiability of a formula: $(\psi \wedge D_i) \models_{\mathcal{T}} \bot \iff \psi \models_{\mathcal{T}} \bot$.

(ii) The conjunction of φ with all the $\mathcal{T}$-lemmas $D_1, \ldots, D_k$ is propositionally unsatisfiable: $\mathcal{T}2\mathcal{B}(\varphi \wedge \bigwedge_{i=1}^{n} D_i) \models \bot$.

Fact (i) is self-evident. Fact (ii) is the termination condition of all lazy SMT tools when the input formula is $\mathcal{T}$-unsatisfiable (lines 15–17 of Figure 1.1 on page 19 [4]).

Example 3.5. Consider again formula (3.1) of Example 3.2. In order to decide its unsatisfiability, MATHSAT generates the following set of $\mathcal{LA}(\mathbb{Z})$-lemmas:

$$\{(\neg(x = 1) \vee \neg(x = 0)), (\neg(y = 2) \vee \neg(y < 0)), (\neg(y = 1) \vee \neg(y < 0))\}. \tag{3.6}$$

Notice that they are all $\mathcal{LA}(\mathbb{Z})$-valid (fact (i)). Then, the Boolean skeleton of (3.1) is conjoined with the Boolean skeleton of these $\mathcal{LA}(\mathbb{Z})$-lemmas, resulting in the following propositional formula:

$$\begin{aligned}(B_0 \vee \neg B_1 \vee A_1) \wedge (B_0 \vee B_1 \vee A_2) \wedge (\neg B_0 \vee B_1 \vee A_2) \wedge (\neg A_2 \vee B_2) \wedge \\ (\neg A_1 \vee B_3) \wedge B_4 \wedge (A_2 \vee B_5) \wedge (B_6 \vee \neg A_1) \wedge B_7 \wedge \\ (\neg B_1 \vee \neg B_0) \wedge (\neg B_6 \vee \neg B_4) \wedge (\neg B_2 \vee \neg B_4),\end{aligned} \tag{3.7}$$

where:

$$\begin{aligned} B_0 &\stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}(x = 0) & B_4 &\stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}(y < 0) \\ B_1 &\stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}(x = 1) & B_5 &\stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}(x - y = 4) \\ B_2 &\stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}(y = 1) & B_6 &\stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}(y = 2) \\ B_3 &\stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}(x + y > 3) & B_7 &\stackrel{\text{def}}{=} \mathcal{T}2\mathcal{B}(x \geq 0). \end{aligned}$$

[4] This can be seen by noticing that $\mathcal{T}$-backjumping on a theory-conflict clause D_i produces an analogous effect as re-invoking DPLL on $\varphi^p \wedge \mathcal{T}2\mathcal{B}(D_i)$, whilst theory propagation on a deduction $\{l_1, \ldots, l_k\} \models_{\mathcal{T}} l$ can be seen as a form on unit propagation on the theory-deduction clause $\mathcal{T}2\mathcal{B}(\bigvee_i \neg l_i \vee l)$.

It is easy to see that (3.7) is unsatisfiable (fact (ii)). ◊

Fact (ii) holds also for those SMT tools which learn mixed Boolean+theory-clauses $F_1, \ldots, F_n$ (instead of $\mathcal{T}$-lemmas), obtained from the $\mathcal{T}$-lemmas $D_1, \ldots, D_n$ by backward traversal of the implication graph. In fact, in this case, $\mathcal{T}2\mathcal{B}(\varphi \wedge \bigwedge_{i=1}^{n} F_i) \models \bot$ holds. Since $\varphi \wedge \bigwedge_{i=1}^{n} D_i \models \bigwedge_{i=1}^{n} F_i$, because of the way the F_i's are built, [5] Fact (ii) holds.

Some SMT tools implement theory-propagation in a slightly different way (e.g. BarceLogic [BNO+08a]). If $l_1, \ldots, l_n \models_{\mathcal{T}} l$, instead of learning the $\mathcal{T}$-lemma $\neg l_1 \vee \ldots \vee \neg l_n \vee l$ and unit-propagating l on it, they simply propagate the value of l, without learning any clause. Only if such propagation leads to a conflict later in the search, the theory-deduction clause is learned and used for conflict-analysis. The validity of fact (ii) is not affected by this optimization, because only the $\mathcal{T}$-lemmas used during conflict analysis are needed for it to hold [NOT06].

Overall, in all variants of the on-line lazy SMT schema (§1.3.1 on page 18), the embedded DPLL engine builds –either explicitly or implicitly– a resolution refutation of the Boolean skeleton of the conjunction of the original clauses and the $\mathcal{T}$-lemmas returned by the $\mathcal{T}$-solver. Thus fact (ii) holds.

3.2.2 Extracting SMT cores by Lifting Theory Lemmas

Facts (i) and (ii) discussed in §3.2.1 suggest a new approach to the generation of unsatisfiable cores for SMT. The main idea is that if the theory lemmas used during the SMT search are lifted into Boolean clauses, then the unsat core can be extracted by a purely propositional core extractor. Therefore, we call this technique the *Lemma-Lifting approach*.

[5] Each clause $\mathcal{T}2\mathcal{B}(F_i)$ is obtained by resolving the clause $\mathcal{T}2\mathcal{B}(D_i)$ with clauses in $\mathcal{T}2\mathcal{B}(\varphi \wedge \bigwedge_{j=1}^{i-1} F_j)$, so that $\mathcal{T}2\mathcal{B}(\varphi \wedge \bigwedge_{j=1}^{i-1} F_j \wedge D_i) \models \mathcal{T}2\mathcal{B}(F_i)$. Thus, by induction, $\mathcal{T}2\mathcal{B}(\varphi \wedge \bigwedge_{i=1}^{n} D_i) \models \mathcal{T}2\mathcal{B}(\bigwedge_{i=1}^{n} F_i)$, so that $\varphi \wedge \bigwedge_{i=1}^{n} D_i \models \bigwedge_{i=1}^{n} F_i$.

$\langle$SatValue, ClauseSet$\rangle$ $\mathcal{T}$-unsat-core (ClauseSet φ)
1. // φ is $\{C_1, \ldots, C_n\}$
2. **if** lazy-smt-solver (φ) == sat **then**
3. **return** $\langle$sat, $\emptyset\rangle$
4. **else**
5. // $D_1, \ldots, D_k$ are the $\mathcal{T}$-lemmas stored by lazy-smt-solver
6. ψ^p = boolean-unsat-core($\mathcal{T}2\mathcal{B}(\{C_1, \ldots, C_n, D_1, \ldots, D_k\})$)
7. // ψ^p is $\mathcal{T}2\mathcal{B}(\{C'_1, \ldots, C'_m, D'_1, \ldots, D'_j\})$
8. **return** $\langle$unsat, $\{C'_1, \ldots, C'_m\}\rangle$
9. **end if**

Figure 3.2: Schema of the $\mathcal{T}$-unsat-core procedure.

The algorithm is presented in Figure 3.2. The procedure $\mathcal{T}$-unsat-core receives as input a set of clauses $\varphi \stackrel{\text{def}}{=} \{C_1, \ldots, C_n\}$ and it invokes on it a lazy SMT($\mathcal{T}$) tool lazy-smt-solver, which is instructed to *store* somewhere the $\mathcal{T}$-lemmas returned by the $\mathcal{T}$-solver, namely $D_1, \ldots, D_k$. If lazy-smt-solver returns sat, then the whole procedure returns sat. Otherwise, the Boolean abstraction of $\{C_1, \ldots, C_n,\ D_1, \ldots, D_k\}$, which is inconsistent because of fact (ii), is fed to an external tool boolean-unsat-core, which is able to return the Boolean unsat core ψ^p of the input. By construction, ψ^p is the Boolean skeleton of a clause set $\{C'_1, \ldots, C'_m, D'_1, \ldots, D'_j\}$ such that $\{C'_1, \ldots, C'_m\} \subseteq \{C_1, \ldots, C_n\}$ and $\{D'_1, \ldots, D'_j\} \subseteq \{D_1, \ldots, D_k\}$. As ψ^p is unsatisfiable, then $\{C'_1, \ldots, C'_m, D'_1, \ldots, D'_j\}$ is $\mathcal{T}$-unsatisfiable. By fact (i), the $\mathcal{T}$-valid clauses $D'_1, \ldots, D'_j$ have no role in the $\mathcal{T}$-unsatisfiability of $\{C'_1, \ldots, C'_m, D'_1, \ldots, D'_j\}$, so that they can be thrown away, and the procedure returns unsat and the $\mathcal{T}$-unsatisfiable core $\{C'_1, \ldots, C'_m\}$.

Notice that the resulting $\mathcal{T}$-unsatisfiable core is not guaranteed to be minimal, even if boolean-unsat-core returns minimal Boolean unsatisfiable cores. In fact, it might be the case that $\{C'_1, \ldots, C'_m\} \backslash \{C'_i\}$ is $\mathcal{T}$-unsatisfiable for some C'_i even though $\mathcal{T}2\mathcal{B}(\{C'_1, \ldots, C'_m\} \setminus \{C'_i\})$ is satisfiable, because all truth assignments μ^p satisfying the latter are such that $\mathcal{B}2\mathcal{T}(\mu^p)$ is

$\mathcal{T}$-unsatisfiable.

Example 3.6. Consider the unsatisfiable SMT formula φ on $\mathcal{LA}(\mathbb{Z})$:

$$\begin{aligned}\varphi \;\equiv\; & (x = 0 \vee x = 1) \wedge (\neg(x = 0) \vee x = 1) \wedge (x = 0 \vee \neg(x = 1)) \wedge \\ & (\neg(x = 0) \vee \neg(x = 1))\end{aligned}$$

and its Boolean skeleton $\mathcal{T}2\mathcal{B}(\varphi)$:

$$\mathcal{T}2\mathcal{B}(\varphi) \equiv (B_0 \vee B_1) \wedge (\neg B_0 \vee B_1) \wedge (B_0 \vee \neg B_1) \wedge (\neg B_0 \vee \neg B_1).$$

Then, $\mathcal{T}2\mathcal{B}(\varphi)$ is a minimal Boolean unsatisfiable core of itself, but φ is not a minimal core in $\mathcal{LA}(\mathbb{Z})$, since the last clause is valid in this theory, and hence it can be safely dropped. $\Diamond$

The procedure can be implemented very simply by modifying the SMT solver so that to store the $\mathcal{T}$-lemmas and by interfacing it with some state-of-the-art Boolean unsat-core extractor used as an external black-box device. Moreover, if the SMT solver can provide the set of all $\mathcal{T}$-lemmas as output, then the whole procedure may reduce to a control device interfacing with both the SMT solver and the Boolean core extractor as black-box external devices.

Remark 3.7. *Notice that here* **storing** *the $\mathcal{T}$-lemmas does not mean* **learning** *them, that is, the SMT solver is not required to add the $\mathcal{T}$-lemmas to the formula during the search. Instead, it is for instance sufficient to store them in some ad-hoc data structure, or even to dump them to a file. This causes no overhead to the Boolean search in the SMT solver, and imposes no constraint on the lazy strategy adopted (e.g., permanent/temporary learning, usage of mixed Boolean+theory conflict clauses, etc.).*

Example 3.8. Once again, consider formula (3.1) of Example 3.2, and the corresponding formula (3.7) of Example 3.5, which is the Boolean skeleton

of (3.1) and the $\mathcal{LA}(\mathbb{Z})$-lemmas (3.6) found by MATHSAT during search. In the Lemma-Lifting approach, (3.7) is given as input to an external Boolean unsat-core device. The resulting propositional unsatisfiable core is:

$$\{(B_0 \vee \neg B_1 \vee A_1), (B_0 \vee B_1 \vee A_2), (\neg B_0 \vee B_1 \vee A_2), (\neg A_2 \vee B_2), B_4,$$
$$(B_6 \vee \neg A_1), (\neg B_1 \vee \neg B_0), (\neg B_6 \vee \neg B_4), (\neg B_2 \vee \neg B_4)\},$$

which corresponds (via $\mathcal{B}2\mathcal{T}$) to:

$$\{(x = 0 \vee \neg(x = 1) \vee A_1), (x = 0 \vee x = 1 \vee A_2), (\neg(x = 0) \vee x = 1 \vee A_2),$$
$$(\neg A_2 \vee y = 1), B_4, (y = 2 \vee \neg A_1),$$
$$(\neg(x = 1) \vee \neg(x = 0)), (\neg(y = 2) \vee \neg(y < 0)), (\neg(y = 1) \vee \neg(y < 0))\}.$$

Since the last three clauses are included in the $\mathcal{LA}(\mathbb{Z})$-lemmas, and thus are $\mathcal{LA}(\mathbb{Z})$-valid, they are eliminated. The resulting core only consists of the first 6 clauses. In this case, the core turns out to be minimal, and is identical modulo reordering to that computed by MATHSAT with proof-tracing (see Example 3.2). $\Diamond$

As observed at the end of the previous section, our technique works also if the SMT tool learns mixed Boolean+theory clauses (provided that the original $\mathcal{T}$-lemmas are stored), or uses the lazy theory deduction of [NOT06]. Moreover, it works also if $\mathcal{T}$-lemmas contain *new atoms* (i.e. atoms that do not appear in φ), as in [FJOS03, BNOT06], since both Facts (ii) and (i) hold also in that case.

As a side observation, we remark that the technique works also for the *per-constraint-encoding* eager SMT approach of [GSZ+98, SSB02]. In the eager SMT approach, the input $\mathcal{T}$-formula φ is translated into an equi-satisfiable Boolean formula, and a SAT solver is used to check its satisfiability. With per-constraint-encoding of [GSZ+98, SSB02], the resulting Boolean formula is the conjunction of the Boolean skeleton φ^p of φ and

a formula $\varphi^{\mathcal{T}}$ which is the Boolean skeleton of the conjunction of some $\mathcal{T}$-valid clauses. Therefore, $\varphi^{\mathcal{T}}$ plays the role of the $\mathcal{T}$-lemmas of the lazy approach, and our approach still works. This idea falls out of the scope of this thesis, and is not expanded further.

3.2.3 Discussion

Despite its simplicity, the proposed approach is appealing for several reasons.

First, it is extremely simple to implement. The building of unsat cores is demanded to an external device, which is fully decoupled from the internal DPLL-based enumerator. Therefore, there is no need of implementing any internal unsat-core constructor nor to modify the embedded Boolean device. Every possible external device can be interfaced in a plug-and-play manner by simply exchanging a couple of DIMACS files [6].

Second, the approach is fully compatible with optimizations carried out by the core extractor at the Boolean level: every original clause which the Boolean unsat-core device is able to drop, is also dropped in the final formula. Notably, this involves also Boolean unsat-core techniques which could be very difficult to adapt to the SMT setting (and to implement within an SMT solver), such as the ones based on genetic algorithms [ZLS06].

Third, it benefits for free from the research on propositional unsat-core extraction, since it is trivial to update: once some novel, more efficient or more effective Boolean unsat-core device is available, it can be used in a plug-and-play way. This does not require modifying the DPLL engine embedded in the SMT solver.

One may remark that, in principle, if the number of $\mathcal{T}$-lemmas generated by the $\mathcal{T}$-solver were huge, the storing of all $\mathcal{T}$-lemmas might cause

[6] DIMACS is a standard format for representing Boolean CNF formulae.

memory-exhaustion problems or the generation of Boolean formulae which are too big to be handled by the Boolean unsat-core extractor. In practice, however, this is not a real problem. In fact, even the hardest SMT formulae at the reach of current lazy SMT solvers rarely need generating more than 10^5 $\mathcal{T}$-lemmas, which require reasonable amount of memory to store, and are well at the reach of current Boolean unsat-core extractors (which can handle formulae in the order of $10^6 - 10^7$ clauses.) For instance, notice that the default choice in MATHSAT is to learn all $\mathcal{T}$-lemmas permanently anyway, and we have never encountered problems due to this fact. Intuitively, unlike with plain SAT, in lazy SMT the computational effort is typically dominated by the search in the theory $\mathcal{T}$, so that the number of clauses that can be stored with a reasonable amount of memory, or which can be fed to a SAT solver, is typically much bigger than the number of calls to the $\mathcal{T}$-solver which can overall be accomplished within a reasonable amount of time.

Like with the other SMT unsat-core techniques adopted by current SMT solvers, also with our novel approach the resulting $\mathcal{T}$-unsatisfiable core is not guaranteed to be minimal, even if boolean-unsat-core returns minimal Boolean unsatisfiable cores. However, with the Lemma-Lifting technique it is possible to perform all the reductions that can be done by considering only the Boolean skeleton of the formula. Although this is in general not enough to guarantee minimality, it is still a very significant gain, as we shall show in the next section.

3.3 Empirical Evaluation

We carried out an extensive experimental evaluation of the the Lemma-Lifting approach. We implemented the approach within MATHSAT, which has been extended with an interface for external Boolean unsatisfiable core

extractors (UCE) to exchange Boolean formulae and relative cores in form of files in DIMACS format.

We have tried eight different external UCEs, namely AMUSE [OMA+04], PICOSAT [Bie08b], EUREKA [DHN06], MUNSAT [vMW08], MUP [Hua05], TRIMMER [GKS08], ZCHAFF [ZM03], and the tool presented in [ZLS06] (called GENETIC here). All these tools explicitly target core size reduction (or minimality), with the exception of PICOSAT, which was conceived for speeding up core generation, with no claims of minimality. In fact, PICOSAT turned out to be both the fastest and the least effective in reducing the size of the cores. Therefore, we adopted it as our baseline choice, as it is the ideal starting point for evaluating the trade-off between efficiency (in execution time) and effectiveness (in core size reduction).

All the experiments have been performed on a subset of the SMT-LIB [RT06] benchmarks. We used a total of 561 $\mathcal{T}$-unsatisfiable problems, taken from the QF_UF (126), QF_IDL (89), QF_RDL (91), QF_LIA (135) and QF_LRA (120) divisions, selected using the same criteria used in the annual SMT competition. In particular, the benchmarks are selected randomly from the available instances in the SMT-LIB, but giving a higher probability to real-world instances, as opposed to randomly generated or handcrafted ones. (See `http://www.smtcomp.org/` for additional details.)

We used a preprocessor to convert the instances into CNF (when required), and in some cases we had to translate them from the SMT language to the native language of a particular SMT solver. [7]

All the tests were performed on 2.66 GHz Intel Xeon machines with 16 GB of RAM running Linux. For each tested instance, the timeout was set to 600 seconds, and the memory limit to 2 GB. For all the Boolean UCEs, we have used the default configurations.

[7] In particular, CVC3 and YICES can compute unsatisfiable cores only if the problems are given in their own native format.

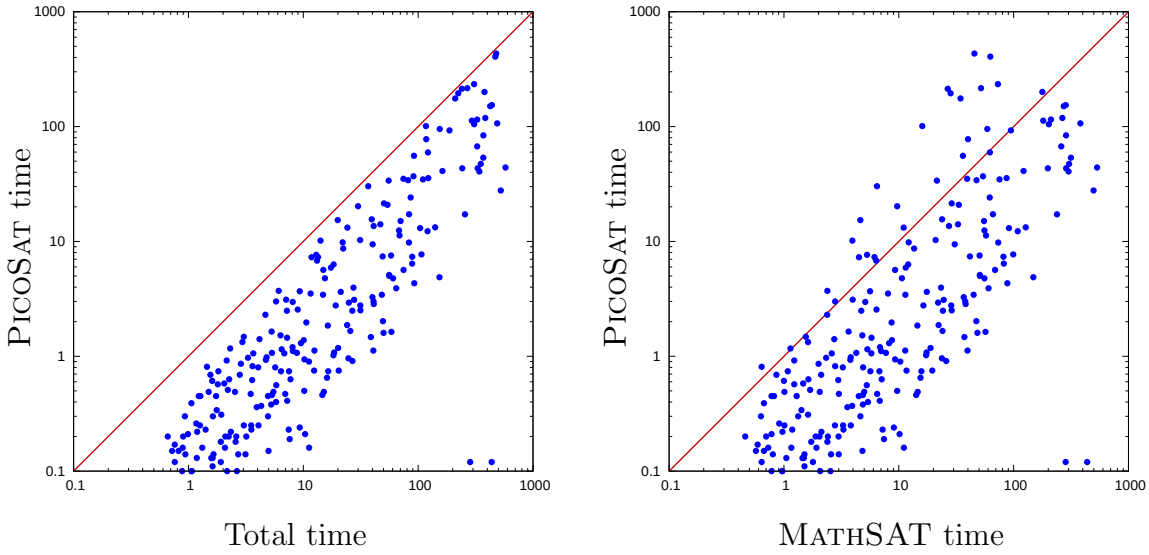


Figure 3.3: Overhead of PicoSat wrt. the total execution time of MathSAT +PicoSat (left) and wrt. the execution time of MathSAT (right).

3.3.1 Costs and effectiveness of unsat-core extraction using PicoSat

The two scatter plots in Figure 3.3 give a first insight on the price that the Lemma-Lifting approach has to pay for running the external UCE. The plot on the left compares the execution time of PicoSat with the total time of MathSAT +PicoSat, whilst the plot on the right shows the comparison of the time of PicoSat against that of MathSAT solving time only. From the two figures, it can be clearly seen that, except for few cases, the time required by PicoSat is much lower or even negligible wrt. MathSAT solving time. We recall that this price is payed only in the case of unsatisfiable benchmarks.

We now analyze our Lemma-Lifting approach with respect to the size of the unsat cores returned. We compare the baseline implementation of our Lemma-Lifting approach, MathSAT +PicoSat , against CVCLite [BT07], [8] Yices and MathSAT +ProofBasedUC (i.e. MathSAT

[8]We tried to use the newer CVC3, but we had some difficulties in the extraction of unsatisfiable cores

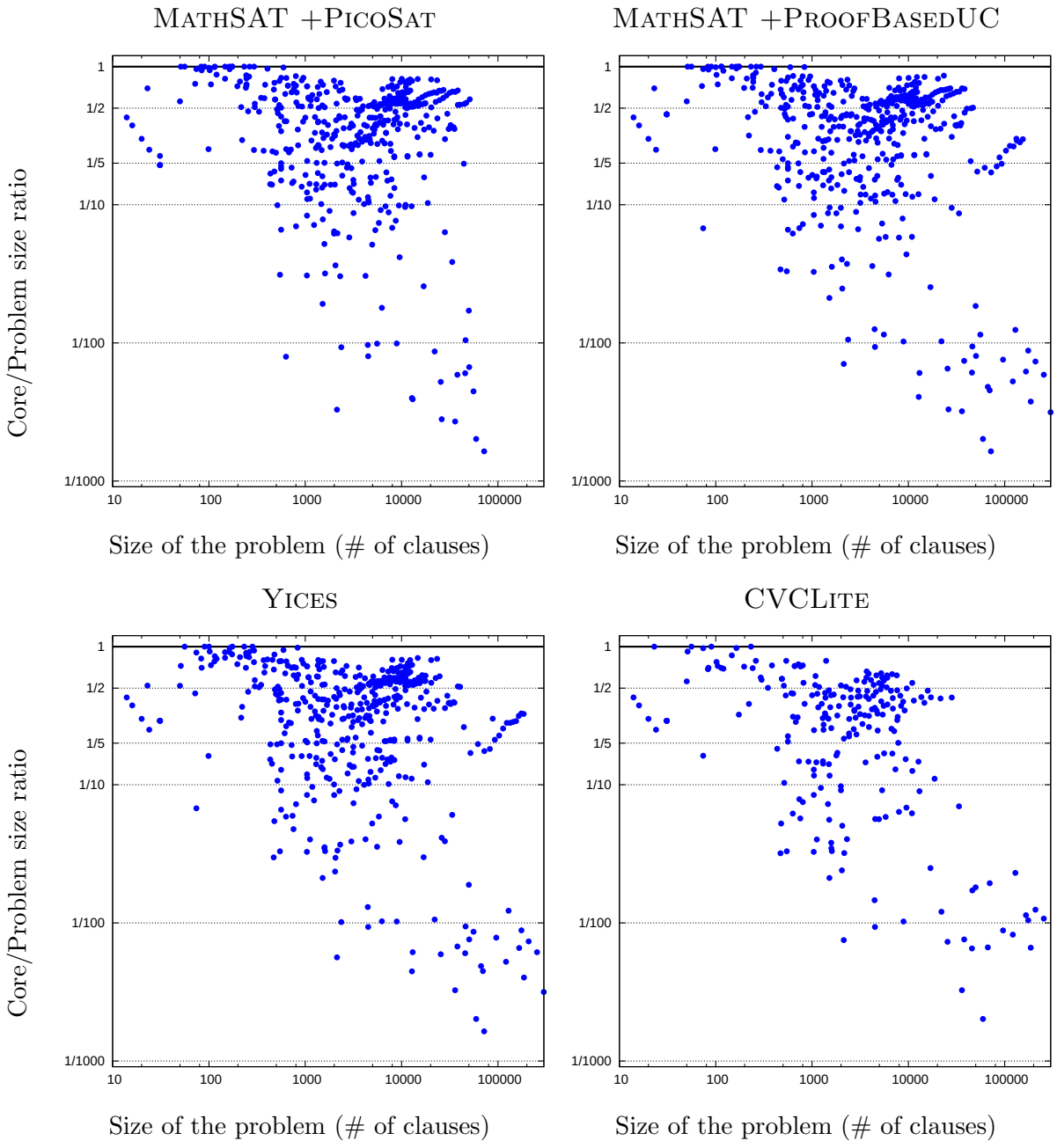
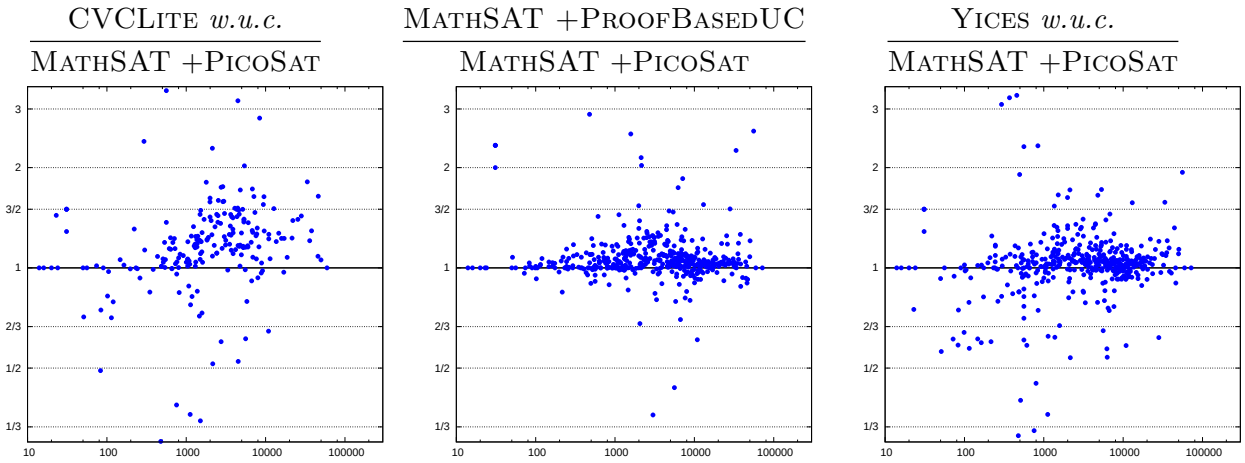


Figure 3.4: Ratio between the size of the original formula and that of the unsat core computed by the various solvers.



core size ratio	1^{st} quartile	median	mean	3^{rd} quartile
CVCLite *w.u.c.* / MathSAT +PicoSat	1.00	1.16	1.33	1.36
MathSAT +ProofBasedUC / MathSAT +PicoSat	1.00	1.03	1.09	1.10
Yices *w.u.c.* / MathSAT +PicoSat	0.97	1.03	1.08	1.09

Figure 3.5: Comparison of the size of the unsat cores computed by MathSAT +PicoSat against those of CVCLite, MathSAT +ProofBasedUC and Yices with unsat cores, with statistics on unsat-core ratios.
Points above the middle line and values greater than 1.00 mean better core quality for MathSAT +PicoSat , and vice versa.

with proof tracing). [9] (Notice that we do not present any comparison in time between the different tools because it is not significant for determining the relative cost of unsat-core computation, since for all tools the time is completely dominated by the solving time, which varies a lot from solver to solver; even within MathSAT, proof production requires setting ad-hoc options, which may result into significantly-different solving times since a different search space is explored.)

Figure 3.4 shows the absolute reduction in size performed by the differ-

with it. Therefore, we reverted to the older CVCLite for the experiments.

[9] CVCLite version `20061231` and Yices version `1.0.19`.

ent solvers: the x-axis displays the size (number of clauses) of the problem, whilst the y-axis displays the ratio between the size of the unsat core and the size of the problem. For instance, a point with y value of 1/10 means that the unsatisfiability is due to only 10% of the problem clauses.

Figure 3.5(top) shows relative comparisons of the data of Figure 3.4. Each plot compares MathSAT +PicoSat with each of the other solvers. Such plots, which we shall call "core-ratio" plots, have the following meaning: the x-axis displays the size (number of clauses) of the problem, whilst the y-axis displays the ratio between the size of the unsat core computed by CVCLite, Yices or MathSAT +ProofBasedUC and that computed by MathSAT +PicoSat . For instance, a point with y value of 1/2 means that the unsat core computed by the current solver is half the size of that computed by MathSAT +PicoSat ; values above 1 mean a smaller core for MathSAT +PicoSat .

In each core-ratio plot, we only consider the instances for which both solvers terminated successfully, since here we are only interested in the size of the cores computed, and not in the execution times. Figure 3.5(bottom) reports statistics about the ratios of the unsat core sizes computed by two different solvers.

The results presented show that, even when using as Boolean UCE PicoSat, which is the least effective in reducing the *size* of the cores, the effectiveness of the baseline version of our Lemma-Lifting approach is slightly better than those of the other (Proof-Based or Assumption-Based) tools.

3.3.2 Impact on costs and effectiveness using different Boolean unsat-core extractors

In this second part of our experimental evaluation we compare the results obtained using different UCE's in terms of costs and effectiveness in reduc-

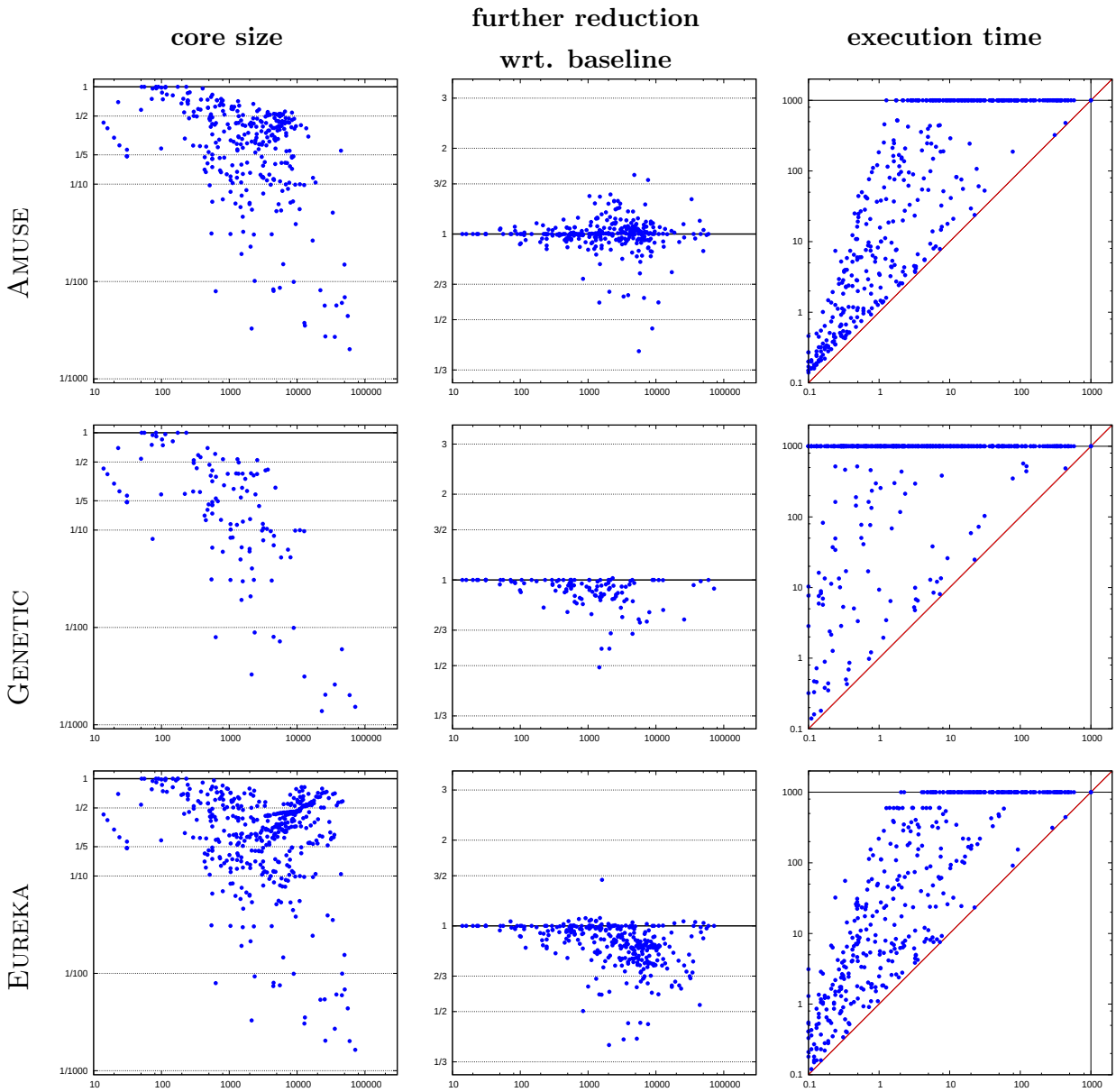


Figure 3.6: Comparison of the core sizes (left), core ratios (middle) and run times (right) using different propositional unsat core extractors. In the core-ratio plots (2^{nd} column), the X-axis represents the size of the problem, and the Y-axis represents the ratio between the size of the cores computed by the two systems: a point above the middle line means better quality for the baseline system. In the scatter plots (3^{rd} column), the baseline system (MATHSAT +PICOSAT) is always on the X-axis.

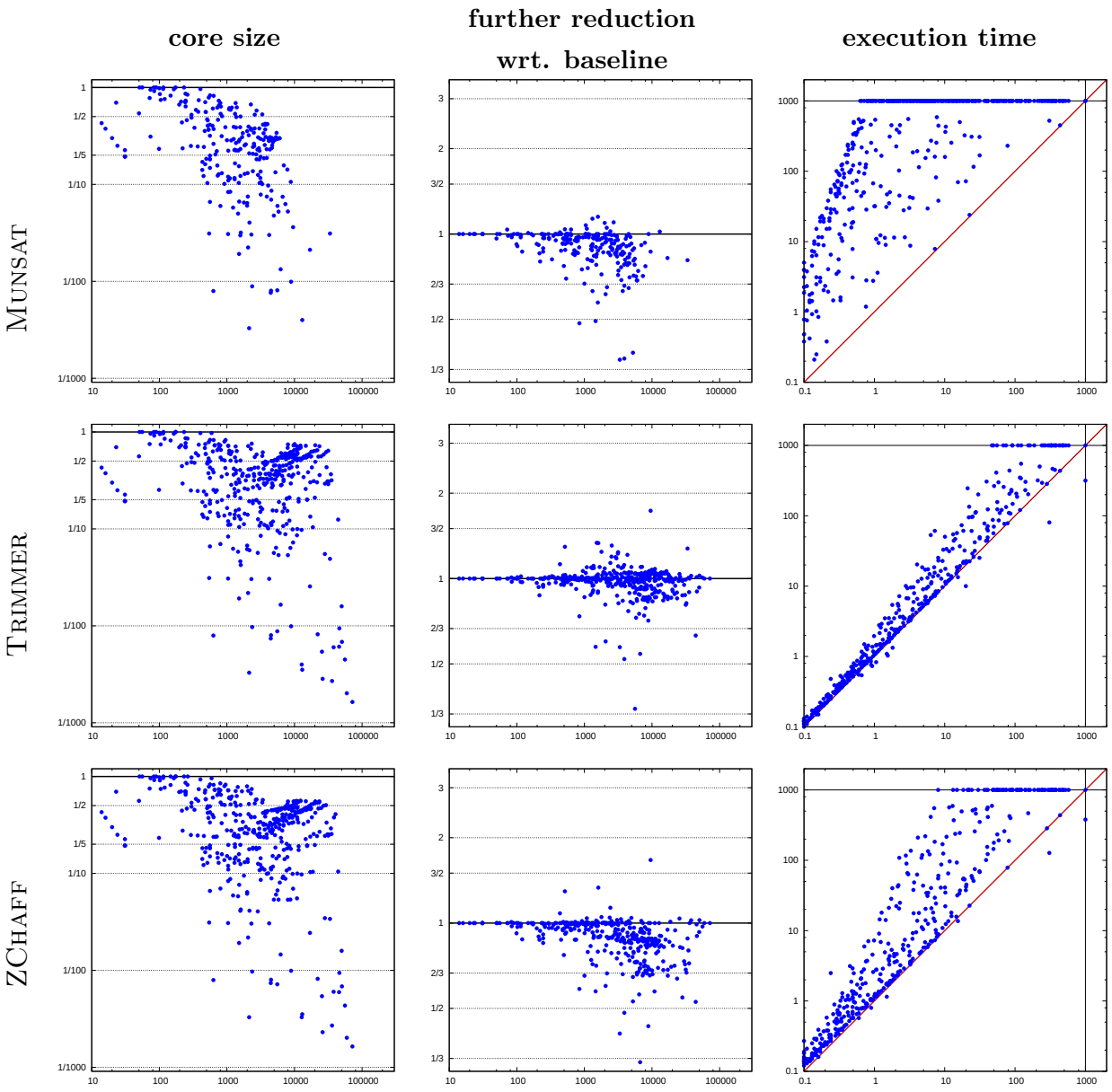
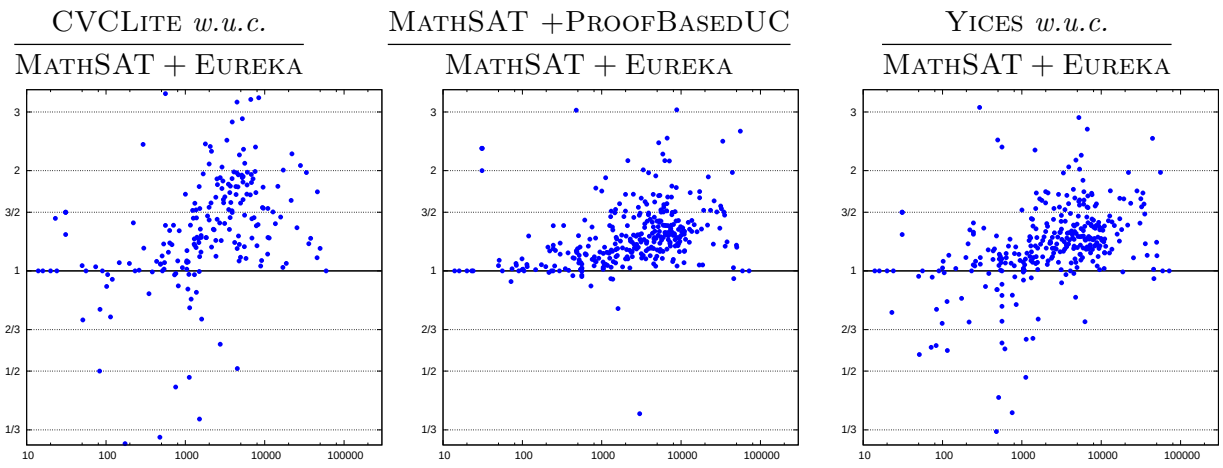


Figure 3.7: Comparison of the core sizes (left), core ratios (middle) and run times (right) using different propositional unsat core extractors (continued).

ing the size of the core. We show that, depending on the UCE used, it is possible to reduce significantly the size of cores, and to trade core quality for speed of execution (and vice versa), with no implementation effort. We compare our baseline configuration MathSAT +PicoSat , against six other configurations, each calling a different propositional UCE.

The results are collected in Figures 3.6-3.7. The first column shows the absolute reduction in size performed by each tool (as in Figure 3.4). The second column shows core-ratio plots comparing each configuration against the baseline one using PicoSat (as in Figure 3.5, with points below 1.00 meaning a better performance of the current configuration). Finally, the scatter plots in the third column compare the execution times (with PicoSat always on the X-axis). We evaluated the six configurations which use, respectively, Amuse [OMA+04], Genetic [ZLS06], Eureka [DHN06], Munsat [vMW08], Trimmer [GKS08], and ZChaff [ZM03], against the baseline configuration, using PicoSat. We also compared with MUP [Hua05], but we had to stop the experiments because of memory exhaustion problems. Looking at the second column, we notice that Eureka, followed by Munsat and ZChaff, seems to be the most effective in reducing the size of the final unsat cores, up to 1/3 the size of those obtained with plain PicoSat. Looking at the third column, we notice that with Genetic, Amuse, Munsat, ZChaff and Eureka, efficiency degrades drastically, and many problems cannot be solved within the timeout. With Trimmer the performance gap is not that dramatic, but still up to an order magnitude slower than the baseline version.

Finally, in Figure 3.8 we compare the effectiveness of MathSAT +Eureka, the most effective extractor in Figures 3.6-3.7, directly with that of the other three solvers, CVCLite, MathSAT +ProofBasedUC and Yices. (Also compare the results with those in Figure 3.5.) The gain in core reduction wrt. previous state-of-the-art SMT core-extraction techniques is



core size ratio	1^{st} quartile	median	mean	3^{rd} quartile
CVCLite *w.u.c.* / MathSAT + Eureka	1.03	1.32	1.55	1.73
MathSAT +ProofBasedUC / MathSAT + Eureka	1.03	1.17	1.27	1.35
Yices *w.u.c.* / MathSAT + Eureka	1.00	1.16	1.28	1.34

Figure 3.8: Ratios of the unsat-core sizes computed by MathSAT +Eureka against those of CVCLite, MathSAT +ProofBasedUC and Yices.
Points above the middle line and values greater than 1.00 mean better core quality for MathSAT +Eureka, and vice versa.

evident.

It is important to notice that, due to our limited know-how, we used the Boolean UCE's in their default configurations. Therefore, we believe that even better results, in terms of both effectiveness and efficiency, could be obtained by means of a more accurate tuning of the parameters of the core extractors.

As a side remark, we notice that the results in Figures 3.6-3.7 have produced as a byproduct an insightful evaluation of the main Boolean unsat-core-generation tools currently available. To this extent, we notice that the performances of MUP [Hua05] and Genetic [ZLS06] seem rather poor; PicoSat [Bie08b] is definitely the fastest tool, though the least effective in

reducing the size of the final core; on the opposite side, Eureka [DHN06] is the most effective in this task, but pays a fee in terms of CPU time; Trimmer [GKS08] represents a good compromise between effectiveness and efficiency.

Chapter 4

Generation of Craig Interpolants

Note. *The material presented in this chapter has already been presented in [CGS08, CGS09c] and [CGS09b].*

Since the seminal paper of McMillan [McM03], the application of (Craig) interpolation is one of the most promising research directions in formal verification. Interpolants have been exploited successfully in the verification of both Boolean/finite-state systems [McM03, CMNQ06, LS06, MS07] as well as infinite-state systems [HJMM04, JM05, JM06, EKS06, McM06, JMX07, JM07, McM08]. As a consequence, in the last few years there has been a lot of research on interpolation procedures and tools, in particular for SMT formulae over several important theories and their combinations [McM05, YM05, SS08, RSS07, KW07, KMZ06, BZM08, JCG08, FGG$^+$09, GKT09].

Quite surprisingly, however, the research on interpolation for SMT has not kept the pace of SMT solving. In fact, most of the approaches to producing interpolants for fragments of first order theories proposed in the last few years [McM05, YM05, RSS07, KW07, KMZ06, JCG08] suffer from a number of problems. Some of the approaches are severely limited in terms of their expressiveness. For instance, the tool described in [JCG08] can only deal with conjunctions of literals, whilst the recent work described in [KW07] can not deal with many useful theories. Furthermore, very few

tools are available [RSS07, McM05, BZM08], and these tools do not seem to scale particularly well. More than to naïve implementation, this appears to be due to the underlying algorithms, that substantially deviate from or ignore choices common in state-of-the-art SMT. For instance, in the domain of linear arithmetic over the rationals ($\mathcal{LA}(\mathbb{Q})$), strict inequalities are encoded in [McM05] as the conjunction of a weak inequality and a disequality; although sound, this choice destroys the structure of the constraints, forces reasoning in the combination of theories $\mathcal{LA}(\mathbb{Q}) \cup \mathcal{EUF}$, requires additional splitting, and ultimately results in a larger search space. Similarly, the fragments of $\mathcal{DL}(\mathbb{Q})$ and $\mathcal{UTVPI}(\mathbb{Q})$ are dealt with by means of a general-purpose algorithm for full $\mathcal{LA}(\mathbb{Q})$, rather than one of the well-known and much faster specialized algorithms. An even more fundamental example is the fact that state-of-the-art SMT reasoners use dedicated algorithms for $\mathcal{LA}(\mathbb{Q})$ [DdM06a], which outperform (sometimes by orders of magnitude) the $\mathcal{LA}(\mathbb{Q})$-decision procedures used in the interpolant-generating tools currently available. Finally, the algorithms proposed in [McM05, YM05] require to use the traditional Nelson-Oppen (N.O.) method [NO79] for computing interpolants in a combination $\mathcal{T}_1 \cup \mathcal{T}_2$ of theories, whereas most current state-of-the-art SMT solvers adopt variants and evolution of the more recent and more flexible Delayed Theory Combination (DTC) approach for theory combination [BBC$^+$06b, BCF$^+$08].

The problem of efficient generation of interpolants in SMT has been addressed only recently. Our work in [CGS08] was the first (to the best of our knowledge) to go in this direction, with the introduction of efficient SMT-based interpolation algorithms for $\mathcal{LA}(\mathbb{Q})$, $\mathcal{DL}$, and combinations of convex theories. Our following work in [CGS09c] covered interpolation for $\mathcal{UTVPI}(\mathbb{Q})$ and $\mathcal{UTVPI}(\mathbb{Z})$. An efficient interpolation algorithm for $\mathcal{EUF}$ was given in [FGG$^+$09]. Finally, another SMT-based method for interpolation in combined theories was recently proposed in [GKT09].

Contributions

We tackle the problem of generating interpolants for SMT problems, fully leveraging the algorithms used in a state of the art SMT solver. In particular, our main contributions are:

1. An interpolation algorithm for $\mathcal{LA}(\mathbb{Q})$ that exploits a variant of the algorithm presented in [DdM06a] (see §2.5), and that is capable of handling the full $\mathcal{LA}(\mathbb{Q})$ – including strict inequalities and disequalities – without the need of theory combination;

2. An algorithm for computing interpolants in $\mathcal{DL}$ – both over the rationals and over the integers – that builds on top of the efficient graph-based decision algorithms given in [CM06, NO05], that ensures that the generated interpolants are still in the $\mathcal{DL}$ fragment of linear arithmetic, and that allows for computing stronger interpolants than the existing algorithms for the full linear arithmetic;

3. An algorithm for computing interpolants in $\mathcal{UTVPI}$ – both over the rationals and over the integers – that builds on an encoding of $\mathcal{UTVPI}$ into $\mathcal{DL}$. The algorithm ensures that the generated interpolants are still in the $\mathcal{UTVPI}$ fragment of linear arithmetic, and allows for computing stronger interpolants than the existing algorithms for the full $\mathcal{LA}$;

4. An algorithm for computing interpolants in a combination $\mathcal{T}_1 \cup \mathcal{T}_2$ of theories based on the Delayed Theory Combination (DTC) method [BBC$^+$06b, BCF$^+$08] (as an alternative to the traditional Nelson-Oppen method), which does not require ad-hoc interpolant combination methods, but exploits the propositional interpolation algorithm for performing the combination of theories;

5. An efficient implementation of all the proposed techniques within MathSAT, and an extensive experimental evaluation on a wide range of benchmarks.

This comprehensive approach advances the state of the art in two main directions: on one side, we show how to extend efficient SMT solving techniques to SMT interpolation, for a wide class of important theories, without paying a substantial price in performance; on the other side, we present an interpolating SMT solver that is able to produce interpolants for a much wider class of problems than its competitors, and, on problems that can be dealt with by other tools, shows dramatic improvements in performance, often by orders of magnitude.

4.1 Background and State of the Art

In the rest of the chapter, we shall use the following notation. If C is a clause and ϕ is a formula, we denote with $C \downarrow \phi$ the clause obtained by removing from C all the literals whose atoms do not occur in ϕ, and with $C \setminus \phi$ that obtained by removing from C all the literals whose atoms do occur in ϕ. If ϕ and ψ are two $\mathcal{T}$-formulae, with $\phi \preceq \psi$ we denote that all uninterpreted (in $\mathcal{T}$) symbols of ϕ occur in ψ.

Definition 4.1 (Craig Interpolant)**.** *Given an ordered pair (A, B) of formulae such that $A \wedge B \models_{\mathcal{T}} \bot$, a* Craig interpolant *(simply "interpolant" hereafter) is a formula I such that:*

(i) $A \models_{\mathcal{T}} I$,

(ii) $I \wedge B \models_{\mathcal{T}} \bot$,

(iii) $I \preceq A$ *and* $I \preceq B$.

Algorithm 4.1: Interpolant generation for SMT($\mathcal{T}$)

1. Generate a resolution proof of unsatisfiability $\mathcal{P}$ for $A \wedge B$.
2. For every $\mathcal{T}$-lemma $\neg\eta$ occurring in $\mathcal{P}$, generate an interpolant $I_{\neg\eta}$ for $(\eta \setminus B, \eta \downarrow B)$.
3. For every input clause C in $\mathcal{P}$, set $I_C \stackrel{\text{def}}{=} C \downarrow B$ if $C \in A$, and $I_C \stackrel{\text{def}}{=} \top$ if $C \in B$.
4. For every inner node C of $\mathcal{P}$ obtained by resolution from $C_1 \stackrel{\text{def}}{=} p \vee \phi_1$ and $C_2 \stackrel{\text{def}}{=} \neg p \vee \phi_2$, set I_C to $I_{C_1} \vee I_{C_2}$ if p does not occur in B, and to $I_{C_1} \wedge I_{C_2}$ otherwise.
5. Output $I_\perp$ as an interpolant for (A, B).

The use of interpolation in formal verification has been introduced by McMillan in [McM03] for purely-propositional formulae, and it was subsequently extended to handle $SMT(\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q}))$ formulae in [McM05]. The technique is based on earlier work by Pudlák [Pud97], where two interpolant-generation algorithms are described: one for computing interpolants for propositional formulae from resolution proofs of unsatisfiability, and one for generating interpolants for conjunctions of (weak) linear inequalities in $\mathcal{LA}(\mathbb{Q})$. An interpolant for a pair (A, B) of CNF formulae is constructed from a resolution proof of unsatisfiability of $A \wedge B$ (see Definition 3.1 on page 81). The algorithm works by computing a formula I_C for each clause in the resolution refutation, such that the formula $I_\perp$ associated to the empty root clause is the computed interpolant. The schema of the algorithm is shown in Algorithm 4.1.

Example 4.2. Consider the following two formulae in $\mathcal{LA}(\mathbb{Q})$:

$$\begin{aligned} A &\stackrel{\text{def}}{=} (p \vee (0 \leq x_1 - 3x_2 + 1)) \wedge (0 \leq x_1 + x_2) \wedge (\neg q \vee \neg(0 \leq x_1 + x_2)) \\ B &\stackrel{\text{def}}{=} (\neg(0 \leq x_3 - 2x_1 - 3) \vee (0 \leq 1 - 2x_3)) \wedge (\neg p \vee q) \wedge \\ &\quad (p \vee (0 \leq x_3 - 2x_1 - 3)) \end{aligned}$$

The uninterpreted symbols in A are $\{p, q, x_1, x_2\}$, and those in B are $\{p, q, x_1, x_3\}$. Therefore, the only uninterpreted symbols that can occur in an interpolant for (A, B) are p, q and x_1.

Figure 4.2(a) shows a resolution proof of unsatisfiability for $A \wedge B$, in which the clauses from A have been underlined. The proof contains the following $\mathcal{LA}(\mathbb{Q})$-lemma (displayed in boldface):

$$\neg(0 \leq x_1 - 3x_2 + 1) \vee \neg(0 \leq x_1 + x_2) \vee \neg(0 \leq x_3 - 2x_1 - 3) \vee \neg(0 \leq 1 - 2x_3).$$

Figure 4.2(b) shows, for each clause Θ_i in the proof, the formula I_{Θ_i} generated by Algorithm 4.1. For the $\mathcal{LA}(\mathbb{Q})$-lemma, it is easy to see that $(0 \leq 4x_1 + 1)$ is an interpolant for $((0 \leq x_1 - 3x_2 + 1) \wedge (0 \leq x_1 + x_2), (0 \leq x_3 - 2x_1 - 3) \wedge (0 \leq 1 - 2x_3))$ as required by Step 2 of the algorithm. (We will show how to obtain this interpolant in Example 4.3.) Therefore, $I_\perp \stackrel{\text{def}}{=} (p \vee (0 \leq 4x_1 + 1)) \wedge \neg q$ is an interpolant for (A, B). $\Diamond$

Algorithm 4.1 can be applied also when A and B are not in CNF. In this case, it suffices to pre-convert them into CNF by using disjoint sets of auxiliary Boolean atoms in the usual way [McM05].

Notice that Step 2. of the algorithm is the only part which depends on the theory $\mathcal{T}$, so that the problem of interpolant generation in SMT($\mathcal{T}$) reduces to that of finding interpolants for $\mathcal{T}$-lemmas. To this extent, in [McM05] McMillan gives a set of rules for constructing interpolants for $\mathcal{T}$-lemmas in the theory of $\mathcal{EUF}$, that of weak linear inequalities $(0 \leq t)$ in $\mathcal{LA}(\mathbb{Q})$, and their combination. Linear equalities $(0 = t)$ can be reduced to conjunctions $(0 \leq t) \wedge (0 \leq -t)$ of inequalities. Thanks to the combination of theories, also strict linear inequalities $(0 < t)$ can be handled in $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ by replacing them with the conjunction $(0 \leq t) \wedge (0 \neq t)$,[1] but this solution can be very inefficient.

[1]The details are not given in [McM05]. One possible way of doing this is to rewrite $(0 \neq t)$ as $(y = t) \wedge (z = 0) \wedge (z \neq y)$, z and y being fresh variables.

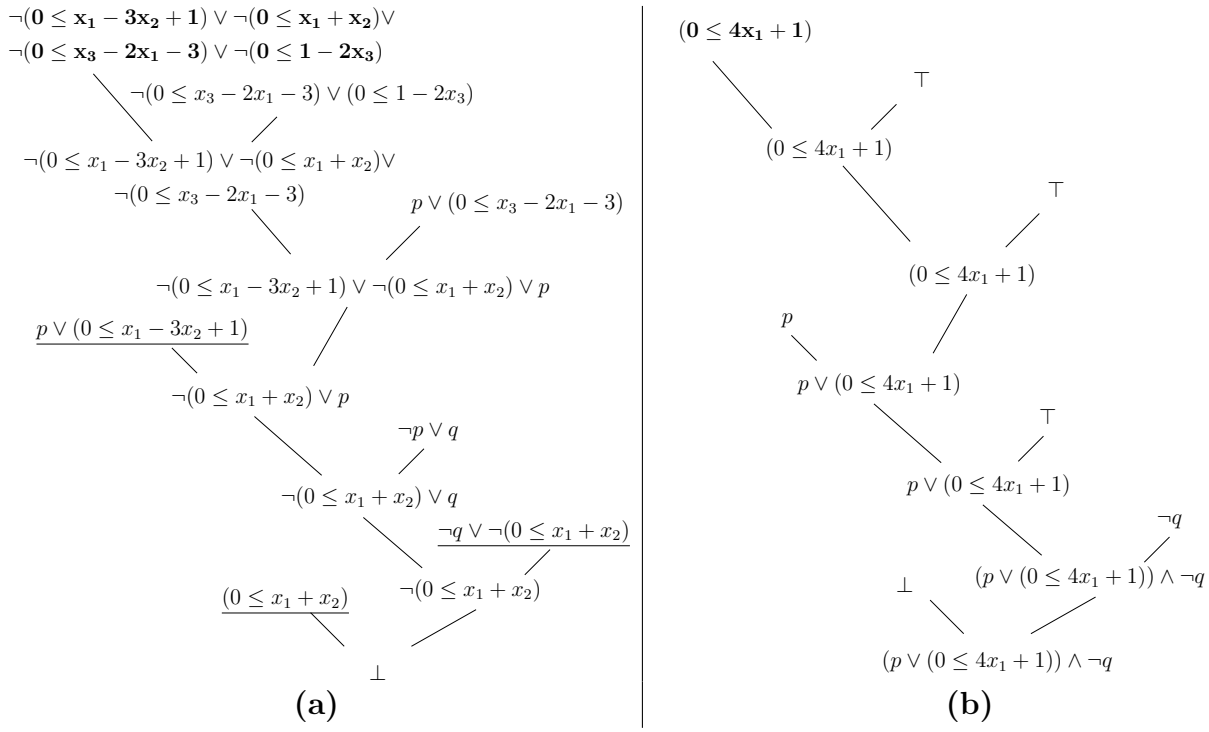


Figure 4.2: Resolution proof of unsatisfiability (a) and interpolant (b) for the pair (A, B) of formulae of Example 4.2. In the tree on the left, $\mathcal{T}$-lemmas are displayed in boldface, and clauses from A are underlined.

An alternative algorithm for computing interpolants in $\mathcal{EUF}$, built on top of the congruence closure algorithm typically used by efficient $\mathcal{EUF}$-solvers in SMT (see §2.4), was recently described in [FGG+09].

The combination $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ can also be used to compute interpolants for other theories, such as those of lists, arrays, sets and multisets [KMZ06].

In [McM05], interpolants in the combined theory $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ are obtained by means of ad-hoc combination rules. The work in [YM05], instead, presents a method for generating interpolants for $\mathcal{T}_1 \cup \mathcal{T}_2$ using the interpolant-generation procedures of $\mathcal{T}_1$ and $\mathcal{T}_2$ as black-boxes, using the N.O. approach (see §1.5.1 on page 31).

Also the method of [RSS07] allows for computing interpolants in $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$. Its peculiarity is that it is not based on unsatisfiability proofs. Instead, it generates interpolants in $\mathcal{LA}(\mathbb{Q})$ by solving a system of constraints

using an off-the-shelf Linear Programming (LP) solver. The method allows both weak and strict inequalities. Extension to uninterpreted functions is achieved by means of reduction to $\mathcal{LA}(\mathbb{Q})$ using a hierarchical calculus [SS08]. The algorithm works only with conjunctions of atoms, although in principle it could be integrated in Algorithm 4.1 to generate interpolants for $\mathcal{T}$-lemmas in $\mathcal{LA}(\mathbb{Q})$. As an alternative, the authors show in [RSS07] how to generate interpolants for formulae that are in Disjunctive Normal Form (DNF).

Another different approach is explored in [KW07]. There, the authors use the *eager* SMT approach to encode the original SMT problem into an equisatisfiable propositional problem, for which a propositional proof of unsatisfiability is generated. This proof is later "lifted" to the original theory, and used to generate an interpolant in a way similar to Algorithm 4.1. At the moment, the approach is however limited to the theory of equality only (without uninterpreted functions).

When moving from $\mathcal{LA}(\mathbb{Q})$ to $\mathcal{LA}(\mathbb{Z})$, interpolation becomes significantly harder. The only known algorithm for computing interpolants in the full $\mathcal{LA}(\mathbb{Z})$ is based on quantifier elimination [KMZ06], which is typically prohibitively expensive. More efficient algorithms for the fragments of $\mathcal{LA}(\mathbb{Z})$ consisting of conjunctions of linear Diophantine equations and disequations and of conjunctions of linear modular equations have been described in [JCG08].

All the above techniques construct *one* interpolant for (A, B). In general, however, interpolants are not unique. In particular, some of them can be better than others, depending on the particular application domain. In [JM05], it is shown how to manipulate proofs in order to obtain stronger interpolants. In [JM06, JM07], instead, a technique to restrict the language used in interpolants is presented and shown to be useful in preventing divergence of techniques based on predicate abstraction.

$$\textsc{LeqEq}\ \frac{0 = t}{0 \leq t} \qquad \textsc{Comb}\ \frac{0 \leq t_1 \qquad 0 \leq t_2}{0 \leq c_1 t_1 + c_2 t_2}\ c_1, c_2 > 0$$

Figure 4.3: $\mathcal{LA}(\mathbb{Q})$-proof rules for a conjunction Γ of equalities and weak inequalities.

One of the most important applications of interpolation in Formal Verification is abstraction refinement [HJMM04, McM06]. In such setting, every input problem ϕ has the form $\phi \stackrel{\text{def}}{=} \phi_1 \wedge \ldots \wedge \phi_n$, and the interpolating solver is asked to compute several interpolants $I_1, \ldots, I_{n-1}$ corresponding to different partitions of ϕ into A_i and B_i, such that

$$\forall i, \quad A_i \stackrel{\text{def}}{=} \phi_1 \wedge \ldots \wedge \phi_i, \quad \text{and} \quad B_i \stackrel{\text{def}}{=} \phi_{i+1} \wedge \ldots \wedge \phi_n. \tag{4.1}$$

Moreover, $I_1, \ldots, I_{n-1}$ should be related by the following:

$$I_i \wedge \phi_{i+1} \models I_{i+1} \tag{4.2}$$

A sufficient condition for (4.2) to hold is that all the I_i's are computed from the same proof of unsatisfiability Π for ϕ [HJMM04].

Interpolants for conjunctions of $\mathcal{LA}(\mathbb{Q})$-literals

We recall the algorithm of [McM05] for computing interpolants from $\mathcal{LA}(\mathbb{Q})$-proofs of unsatisfiability, for conjunctions of equalities and weak inequalities in $\mathcal{LA}(\mathbb{Q})$.

An $\mathcal{LA}(\mathbb{Q})$-*proof rule* R for a conjunction Γ of equalities and weak inequalities is either an element of Γ, or it has the form $\dfrac{P}{\phi}$, where ϕ is an equality or a weak inequality and P is a sequence of proof rules, called the *premises* of R. An $\mathcal{LA}(\mathbb{Q})$-*proof of unsatisfiability* for a conjunction of equalities and weak inequalities Γ is simply a rule in which $\phi \equiv 0 \leq c$ and where c is a negative numerical constant.[2]

[2]In the following, we sometimes write $\bot$ as a synonym of an atom "$0 \leq c$" when c is a negative numerical constant.

Similarly to [McM05], we use the proof rules of Figure 4.3: LeqEq for deriving inequalities from equalities, and Comb for performing linear combinations.[3]

Given an $\mathcal{LA}(\mathbb{Q})$-proof of unsatisfiability P for a conjunction Γ of equalities and weak inequalities partitioned into (A, B), an interpolant I can be computed simply by replacing every atom $0 \leq t$ (resp. $0 = t$) occurring in B with $0 \leq 0$ (resp. $0 = 0$) in each leaf sub-rule of P, and propagating the results: the interpolant is then the single weak inequality $0 \leq t$ at the root of P [McM05].

Example 4.3. Consider the following sets of $\mathcal{LA}(\mathbb{Q})$ atoms:

$$A \stackrel{\text{def}}{=} \{(0 \leq x_1 - 3x_2 + 1), (0 \leq x_1 + x_2)\}$$
$$B \stackrel{\text{def}}{=} \{(0 \leq x_3 - 2x_1 - 3), (0 \leq 1 - 2x_3)\}.$$

An $\mathcal{LA}(\mathbb{Q})$-proof of unsatisfiability P for $A \wedge B$ is the following:

$$\dfrac{\dfrac{1 \cdot (0 \leq x_1 - 3x_2 + 1) \quad 4 \cdot (0 \leq x_1 + x_2)}{1 \cdot (0 \leq 4x_1 + 1)} \qquad \dfrac{2 \cdot (0 \leq x_3 - 2x_1 - 3) \quad 1 \cdot (0 \leq 1 - 2x_3)}{1 \cdot (0 \leq -4x_1 - 5)}}{(0 \leq -4)}$$

By replacing inequalities in B with $(0 \leq 0)$, we obtain the proof P':

$$\dfrac{\dfrac{1 \cdot (0 \leq x_1 - 3x_2 + 1) \quad 4 \cdot (0 \leq x_1 + x_2)}{1 \cdot (0 \leq 4x_1 + 1)} \qquad \dfrac{2 \cdot (0 \leq 0) \quad 1 \cdot (0 \leq 0)}{1 \cdot (0 \leq 0)}}{(0 \leq 4x_1 + 1)}$$

Thus, the interpolant obtained is $(0 \leq 4x_1 + 1)$. $\Diamond$

4.2 From SMT($\mathcal{LA}(\mathbb{Q})$) solving to SMT($\mathcal{LA}(\mathbb{Q})$) interpolation

In this section, we show how to exploit the Dutertre-de Moura algorithm, which represents the state of the art in $\mathcal{LA}(\mathbb{Q})$-decision procedures for SMT

[3] In [McM05] the LeqEq rule is not used in $\mathcal{LA}(\mathbb{Q})$, because the input is assumed to consist only of inequalities.

(see §2.5), to efficiently generate interpolants for sets of $\mathcal{LA}(\mathbb{Q})$ constraints. Interpolation for $SMT(\mathcal{LA}(\mathbb{Q}))$ formulae is then obtained by "plugging" this algorithm into Algorithm 4.1. In §4.2.1 we begin by considering the case in which the input atoms are only equalities and non-strict inequalities. In this case, we only need to show how to generate a proof of unsatisfiability, since then we can use the interpolation rules defined in [McM05]. Then, in §4.2.2 we show how to generate interpolants for problems containing also strict inequalities and disequalities.

4.2.1 Interpolation with non-strict inequalities

In its original formulation, the Dutertre-de Moura algorithm is not immediately suitable for producing interpolants, because in case of inconsistency it does not provide enough information for constructing a detailed proof of unsatisfiability using the rules of Figure 4.3. In particular, referring to its description given in §2.5.1 (page 48 and following), we recall that the algorithm detects an inconsistency when it is not possible to adjust the values of the candidate model β and of the sets $\mathcal{B}$ and $\mathcal{N}$, in order to make β satisfy the bounds on the variables in $\mathcal{B}$, without violating the invariants in (2.2). In such cases, as the bounds on the variables in $\mathcal{N}$ are always satisfied by β, then there is a variable $x_i \in \mathcal{B}$ such that the inconsistency is caused either by the elementary atom $x_i \geq l_i$ or by the atom $x_i \leq u_i$ [DdM06a]; in the first case, [4] a conflict set η is generated as follows:

$$\eta = \{x_j \leq u_j | x_j \in \mathcal{N}^+\} \cup \{x_j \geq l_j | x_j \in \mathcal{N}^-\} \cup \{x_i \geq l_i\}, \tag{4.3}$$

where $(x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j)$ is the row of the current version of the tableau T (2.1) corresponding to x_i, $\mathcal{N}^+$ is $\{x_j \in \mathcal{N} | a_{ij} > 0\}$ and $\mathcal{N}^-$ is $\{x_j \in \mathcal{N} | a_{ij} < 0\}$. However, η is a conflict set only in the sense that it is made inconsistent by (some of) the equations in the tableau T (2.1), i.e. $T \cup \eta \models_{\mathcal{LA}(\mathbb{Q})} \bot$, but

[4] Here we do not consider the second case $x_i \leq u_i$ as it is analogous to the first one.

in general $\eta \not\models_{\mathcal{LA}(\mathbb{Q})} \bot$. Therefore, η alone is not enough for producing a proof of unsatisfiability.

In order to overcome this limitation and to make the algorithm suitable for interpolant generation, we have conceived the following variant of it.

We take as input an arbitrary set of inequalities $l_k \leq \sum_h \hat{a}_{kh}\, y_h$ or $u_k \geq \sum_h \hat{a}_{kh}\, y_h$, and apply an *internal* preprocessing step to obtain a set of equations and a set of elementary atoms as in §2.5.1. In particular, we introduce a "slack" variable s_k for each distinct term $\sum_h \hat{a}_{kh}\, y_h$ occurring in the input inequalities. Then, we replace such term with s_k (thus obtaining $l_k \leq s_k$ or $u_k \geq s_k$) and add an equation $s_k = \sum_h \hat{a}_{kh}\, y_h$. Notice that we introduce a slack variable even for "elementary" inequalities ($l_k \leq y_k$). With this transformation, the initial tableau T (2.1) is:

$$\{s_k = \textstyle\sum_h \hat{a}_{kh}\, y_h\}_k, \tag{4.4}$$

such that $\hat{\mathcal{B}}$ is made of all the slack variables s_k's, $\hat{\mathcal{N}}$ is made of all the original variables y_h's, and the elementary atoms contain only slack variables s_k's.

Then the algorithm proceeds as described in §2.5.1, producing a set η (4.3) in case of inconsistency. In our variant of the algorithm, we can use η to generate a conflict set η', thanks to the following theorem.

Theorem 4.4. *In the set η of* (4.3)*, x_i and all the x_j's are slack variables introduced by our preprocessing step. Moreover, the set $\eta' \stackrel{def}{=} \eta_{\mathcal{N}^+} \cup \eta_{\mathcal{N}^-} \cup \eta_i$ is a conflict set, where*

$$\begin{aligned}
\eta_{\mathcal{N}^+} &\stackrel{def}{=} \{u_k \geq \textstyle\sum_h \hat{a}_{kh}\, y_h | s_k \equiv x_j \textit{ and } x_j \in \mathcal{N}^+\},\\
\eta_{\mathcal{N}^-} &\stackrel{def}{=} \{l_k \leq \textstyle\sum_h \hat{a}_{kh}\, y_h | s_k \equiv x_j \textit{ and } x_j \in \mathcal{N}^-\},\\
\eta_i &\stackrel{def}{=} \{l_k \leq \textstyle\sum_h \hat{a}_{kh}\, y_h | s_k \equiv x_i\}.
\end{aligned}$$

Proof. We consider the case in which η (4.3) is generated from a row $x_i = \sum_{x_j \in \mathcal{N}} a_{ij}\, x_j$ in the tableau T (2.1) such that $\beta(x_i) < l_i$. In [DdM06a]

it is shown that in this case the following facts hold:

$$\forall x_j \in \mathcal{N}^+, \beta(x_j) = u_j, \quad \text{and} \quad \forall x_j \in \mathcal{N}^-, \beta(x_j) = l_j. \tag{4.5}$$

(We recall that $\mathcal{N}^+ = \{x_j \in \mathcal{N} | a_{ij} > 0\}$ and $\mathcal{N}^- = \{x_j \in \mathcal{N} | a_{ij} < 0\}$.) The bounds u_j and l_j can be introduced only by elementary atoms. Since in our variant the elementary atoms contain only slack variables, each x_j must be a slack variable (namely s_k). The same holds for x_i (since its value is bounded by l_i).

Now consider η again. In [DdM06a] it is shown that when a conflict is detected because $\beta(x_i) < l_i$, then the following fact holds:

$$\beta(x_i) = \textstyle\sum_{x_j \in \mathcal{N}^+} a_{ij} u_j + \sum_{x_j \in \mathcal{N}^-} a_{ij} l_j. \tag{4.6}$$

From the i-th row of the tableau T (2.1) we can derive

$$0 \leq \textstyle\sum_{x_j \in \mathcal{N}} a_{ij}\, x_j - x_i. \tag{4.7}$$

If we take each inequality $0 \leq u_j - x_j$ multiplied by the coefficient a_{ij} for all $x_j \in \mathcal{N}^+$, each inequality $0 \leq x_j - l_j$ multiplied by coefficient $-a_{ij}$ for all $x_j \in \mathcal{N}^-$, and the inequality $(0 \leq x_i - l_i)$ multiplied by 1, and we add them to (4.7), then we obtain

$$0 \leq \textstyle\sum_{\mathcal{N}^+} a_{ij}\, u_j + \sum_{\mathcal{N}^-} a_{ij}\, l_j - l_i, \tag{4.8}$$

which by (4.6) is equivalent to $0 \leq \beta(x_i) - l_i$. Thus we have obtained $0 \leq c$ with $c \equiv \beta(x_i) - l_i$, which is strictly lower than zero. Therefore, η is inconsistent under the definitions in T. Since we know that x_i and all the x_j's in η are slack variables, we can replace every x_j (i.e., every s_k) with its corresponding term $\sum_h \hat{a}_{kh}\, y_h$, thus obtaining η', which is thus inconsistent. □

When our variant of the algorithm detects an inconsistency, we construct a proof of unsatisfiability as follows. From the set η of (4.3) we build a conflict set η' by replacing each elementary atom in it with the corresponding

original atom, as shown in Theorem 4.4. We then combine all the atoms in $\eta_{\mathcal{N}^+}$ with repeated applications of the COMB rule: if $u_k \geq \sum_h \hat{a}_{kh}\, y_h$ is the atom corresponding to s_k, we use as coefficient for the COMB the a_{ij} (in the i-th row of the current tableau) such that $s_k \equiv x_j$. Then, we add each of the atoms in $\eta_{\mathcal{N}^-}$ to the previous combination, again using COMB. In this case, the coefficient to use is $-a_{ij}$. Finally, we add the atom in η_i to the combination with coefficient 1.

Corollary 4.5. *The result of the linear combination described above is the atom $0 \leq c$, such that c is a numerical constant strictly lower than zero.*

Proof. Follows immediately by the proof of Theorem 4.4. □

Besides the case just described (and its dual when the inconsistency is due to an elementary atom $x_i \leq u_i$), another case in which an inconsistency can be detected is when two contradictory atoms are asserted: $l_k \leq \sum_h \hat{a}_{kh}\, y_h$ and $u_k \geq \sum_h \hat{a}_{kh}\, y_h$, with $l_k > u_k$. In this case, the proof is simply the combination of the two atoms with coefficient 1.

The extension for handling also equalities like $b_k = \sum_h \hat{a}_{kh}\, y_h$ is straightforward: we simply introduce two elementary atoms $b_k \leq s_k$ and $b_k \geq s_k$ and, in the construction of the proof, we use the LEQEQ rule to introduce the proper inequality.

Finally, notice that the current implementation in MATHSAT is slightly different from what presented here, and significantly more efficient. In practice, η, η' are not constructed in sequence; rather, they are built simultaneously. Moreover, some optimizations are applied to eliminate some slack variables when they are not needed (see also [dMB08b] for more de-

tails about this last point).

Example 4.6. Consider again the two sets of $\mathcal{LA}(\mathbb{Q})$ atoms of Example 4.3:

$$A \stackrel{\text{def}}{=} \{(0 \leq x_1 - 3x_2 + 1), (0 \leq x_1 + x_2\}$$
$$B \stackrel{\text{def}}{=} \{(0 \leq x_3 - 2x_1 - 3), (0 \leq 1 - 2x_3)\}.$$

With our variant of the Dutertre-de Moura algorithm, four "slack" variables are introduced, resulting in the following tableau and elementary constraints:

$$T \stackrel{\text{def}}{=} \begin{cases} s_1 = x_1 - 3x_2 & -1 \leq s_1 \\ s_2 = x_1 + x_2 & 0 \leq s_2 \\ s_3 = x_3 - 2x_1 & 3 \leq s_3 \\ s_4 = -2x_3 & -1 \leq s_4 \end{cases}$$

To detect the inconsistency, the algorithm performs some pivoting steps, resulting in the final tableau T':

$$T' \stackrel{\text{def}}{=} \begin{cases} x_2 = -\frac{1}{12}s_4 - \frac{1}{6}s_3 - \frac{1}{3}s_1 \\ s_2 = -\frac{1}{3}s_4 - \frac{2}{3}s_3 - \frac{1}{3}s_1 \\ x_1 = -\frac{1}{4}s_4 - \frac{1}{2}s_3 \\ x_3 = -\frac{1}{2}s_4 \end{cases}$$

The final values of β are as follows:

$$\begin{array}{llll} \beta(x_1) = \frac{7}{4} & \beta(x_2) = -\frac{1}{12} & \beta(x_3) = \frac{1}{2} & \\ \beta(s_1) = -1 & \beta(s_2) = -\frac{4}{3} & \beta(s_3) = 3 & \beta(s_4) = -1 \end{array}$$

Therefore, the bound $(0 \leq s_2)$ is violated. From the second row of T', the set η and the conflict set η' are computed:

$$\eta \stackrel{\text{def}}{=} \emptyset \cup \{(-1 \leq s_4), (3 \leq s_3), (-1 \leq s_1)\} \cup \{(0 \leq s_2)\}$$
$$\eta' \stackrel{\text{def}}{=} \emptyset \cup \{(0 \leq 1 - 2x_3), (0 \leq x_3 - 2x_1 - 3), (0 \leq x_1 - 3x_2 + 1)\} \cup \{(0 \leq x_1 + x_2)\}$$

The generated proof of unsatisfiability P is:

$$\dfrac{\dfrac{\dfrac{\frac{1}{3}\cdot(0 \leq 1-2x_3) \quad \frac{2}{3}\cdot(0 \leq x_3-2x_1-3)}{1\cdot(0 \leq -\frac{4}{3}x_1-\frac{5}{3})} \quad \frac{1}{3}\cdot(0 \leq x_1-3x_2+1)}{1\cdot(0 \leq -x_1-x_2-\frac{4}{3})} \quad 1\cdot(0 \leq x_1+x_2)}{(0 \leq -\frac{4}{3})}$$

After replacing the inequalities of B with $(0 \leq 0)$ in P, the new proof P' is:

$$\dfrac{\dfrac{\dfrac{\frac{1}{3}\cdot(0 \leq 0) \quad \frac{2}{3}\cdot(0 \leq 0)}{1\cdot(0 \leq 0)} \quad \frac{1}{3}\cdot(0 \leq x_1-3x_2+1)}{1\cdot(0 \leq \frac{1}{3}x_1-x_2+\frac{1}{3})} \quad 1\cdot(0 \leq x_1+x_2)}{(0 \leq \frac{4}{3}x_1+\frac{1}{3})}$$

Thus the computed interpolant is $(0 \leq \frac{4}{3}x_1 + \frac{1}{3})$ (which is equivalent to that of Example 4.3). ◊

4.2.2 Interpolation with strict inequalities and disequalities

Another benefit of the Dutertre-de Moura algorithm is that it can handle strict inequalities directly. Its method is based on the following lemma.

Lemma 4.7 (Lemma 1 in [DdM06a])**.** *A set of linear arithmetic atoms* Γ *containing strict inequalities* $S \stackrel{def}{=} \{0 < t_1, \ldots, 0 < t_n\}$ *is satisfiable if and only if there exists a rational number* $\varepsilon > 0$ *such that* $\Gamma_\varepsilon \stackrel{def}{=} (\Gamma \cup S_\varepsilon) \setminus S$ *is satisfiable, where* $S_\varepsilon \stackrel{def}{=} \{\varepsilon \leq t_1, \ldots, \varepsilon \leq t_n\}$.

The idea of [DdM06a] is that of treating the *infinitesimal parameter* ε symbolically instead of explicitly computing its value. Strict bounds $(x < b)$ are replaced with weak ones $(x \leq b - \varepsilon)$, and the operations on bounds are adjusted to take ε into account.

We extend the same idea to the computation of interpolants. We transform every atom $(0 < t_i)$ occurring in the proof of unsatisfiability into $(0 \leq t_i - \varepsilon)$. Then we compute an interpolant I_ε in the usual way. As a consequence of the rules of [McM05], I_ε is always a single atom. As

shown by the following lemma, if I_ε contains ε, then it must be in the form $(0 \leq t - c\,\varepsilon)$ with $c > 0$, and we can rewrite I_ε into $(0 < t)$.

Theorem 4.8 (Interpolation with strict inequalities)**.** *Let Γ, S, Γ_ε and S_ε be defined as in Lemma 4.7. Let Γ be partitioned into A and B, and let A_ε and B_ε be obtained from A and B by replacing atoms in S with the corresponding ones in S_ε. Let I_ε be an interpolant for $(A_\varepsilon, B_\varepsilon)$. Then:*

- *If $\varepsilon \not\preceq I_\varepsilon$, then I_ε is an interpolant for (A, B).*
- *If $\varepsilon \preceq I_\varepsilon$, then $I_\varepsilon \equiv (0 \leq t - c\,\varepsilon)$ for some $c > 0$, and $I \stackrel{def}{=} (0 < t)$ is an interpolant for (A, B).*

Proof. Since the side condition of the COMB rule ensures that equations are combined only using positive coefficients, and since the atoms introduced in the proof either do not contain ε or contain it with a negative coefficient, if ε appears in I_ε, it must have a negative coefficient.

If ε does not appear in I_ε, then I_ε has been obtained from atoms appearing in A or B, so that I_ε is an interpolant for (A, B).

If ε appears in I_ε, since its value has not been explicitly computed, it can be arbitrarily small, so thanks to Lemma 4.7 we have that $B_\varepsilon \wedge I_\varepsilon \models_{\mathcal{LA}(\mathbb{Q})} \bot$ implies $B \wedge I \models_{\mathcal{LA}(\mathbb{Q})} \bot$.

We can prove that $A \models_{\mathcal{LA}(\mathbb{Q})} I$ as follows. We consider some interpretation μ which is a model for A. Since ε does not occur in A, we can extend μ by setting $\mu(\varepsilon) = \delta$ for some $\delta > 0$ such that μ is a model also for A_ε. As $A_\varepsilon \models_{\mathcal{LA}(\mathbb{Q})} I_\varepsilon$, μ is also a model for I_ε, and hence μ is also a model for I. Thus, we have that $A \models_{\mathcal{LA}(\mathbb{Q})} I$. □

Notice that Theorem 4.8 can be extended straightforwardly to the case in which the interpolant is a conjunction of inequalities.

Thus, in case of strict inequalities, Theorem 4.8 gives us a way for constructing interpolants with no need of expensive theory combination

(as instead was the case in [McM05]). Moreover, thanks to it we can handle also negated equalities $(0 \neq t)$ directly. Suppose our set S of input atoms (partitioned into A and B) is the union of a set S' of equalities and inequalities (both weak and strict) and a set $S^{\neq}$ of disequalities, and suppose that S' is consistent. (If not so, an interpolant can be computed from S'.) Since $\mathcal{LA}(\mathbb{Q})$ is convex, S is inconsistent if and only if exists $(0 \neq t) \in S^{\neq}$ such that $S' \cup \{(0 \neq t)\}$ is inconsistent, that is, such that both $S' \cup \{(0 < t)\}$ and $S' \cup \{(0 > t)\}$ are inconsistent.

Therefore, we pick one element $(0 \neq t)$ of $S^{\neq}$ at a time, and check the satisfiability of $S' \cup \{(0 < t)\}$ and $S' \cup \{(0 > t)\}$. If both are inconsistent, from the two proofs we can generate two interpolants I^- and I^+. We combine I^+ and I^- to obtain an interpolant I for (A, B): if $(0 \neq t) \in A$, then I is $I^+ \vee I^-$; if $(0 \neq t) \in B$, then I is $I^+ \wedge I^-$, as shown by the following Theorem.

Theorem 4.9 (Interpolation for negated equalities)**.** *Let A and B two conjunctions of $\mathcal{LA}(\mathbb{Q})$ atoms, and let $n \stackrel{def}{=} (0 \neq t)$ be one such atom. Let $g \stackrel{def}{=} (0 < t)$ and $l \stackrel{def}{=} (0 > t)$.*
If $n \in A$, then let $A^+ \stackrel{def}{=} A \setminus \{n\} \cup \{g\}$, $A^- \stackrel{def}{=} A \setminus \{n\} \cup \{l\}$, and $B^+ \stackrel{def}{=} B^- \stackrel{def}{=} B$.
If $n \in B$, then let $A^+ \stackrel{def}{=} A^- \stackrel{def}{=} A$, $B^+ \stackrel{def}{=} B \setminus \{n\} \cup \{g\}$, and $B^- \stackrel{def}{=} B \setminus \{n\} \cup \{l\}$.
Assume that $A^+ \wedge B^+ \models_{\mathcal{LA}(\mathbb{Q})} \perp$ and that $A^- \wedge B^- \models_{\mathcal{LA}(\mathbb{Q})} \perp$, and let I^+ and I^- be two interpolants for (A^+, B^+) and (A^-, B^-) respectively, and let

$$I \stackrel{def}{=} \begin{cases} I^+ \vee I^- & \text{if } n \in A \\ I^+ \wedge I^- & \text{if } n \in B. \end{cases}$$

Then I is an interpolant for (A, B).

Proof. We have to prove that:

(i) $A \models_{\mathcal{LA}(\mathbb{Q})} I$

(ii) $B \wedge I \models_{\mathcal{LA}(\mathbb{Q})} \bot$

(iii) $I \preceq A$ and $I \preceq B$.

(i) If $n \in A$, then $A \models_{\mathcal{LA}(\mathbb{Q})} g \vee l$. By hypothesis, we know that $A^+ \models_{\mathcal{LA}(\mathbb{Q})} I^+$ and $A^- \models_{\mathcal{LA}(\mathbb{Q})} I^-$. Then trivially $A \cup \{g\} \models_{\mathcal{LA}(\mathbb{Q})} I^+$ and $A \cup \{l\} \models_{\mathcal{LA}(\mathbb{Q})} I^-$. Therefore $A \cup \{g\} \models_{\mathcal{LA}(\mathbb{Q})} I^+ \vee I^-$ and $A \cup \{l\} \models_{\mathcal{LA}(\mathbb{Q})} I^- \vee I^+$, so that $A \models_{\mathcal{LA}(\mathbb{Q})} I$.

If $n \in B$, then $A^+ \equiv A^- \equiv A$. By hypothesis $A \models_{\mathcal{LA}(\mathbb{Q})} I^+$ and $A \models_{\mathcal{LA}(\mathbb{Q})} I^-$, so that $A \models_{\mathcal{LA}(\mathbb{Q})} I$.

(ii) If $n \in A$, then $B^+ \equiv B^- \equiv B$. By hypothesis $B \wedge I^+ \models_{\mathcal{LA}(\mathbb{Q})} \bot$ and $B \wedge I^- \models_{\mathcal{LA}(\mathbb{Q})} \bot$, so that $B \wedge I \models_{\mathcal{LA}(\mathbb{Q})} \bot$.

If $n \in B$, then $B \models_{\mathcal{LA}(\mathbb{Q})} g \vee l$, so that either $B \rightarrow g$ or $B \rightarrow l$ must hold. By hypothesis we have $B^+ \wedge I^+ \models_{\mathcal{LA}(\mathbb{Q})} \bot$, so that $B \cup \{g\} \wedge I^+ \models_{\mathcal{LA}(\mathbb{Q})} \bot$. If $B \rightarrow g$ holds, then $B \wedge I^+ \models_{\mathcal{LA}(\mathbb{Q})} \bot$, and hence $B \wedge I \models_{\mathcal{LA}(\mathbb{Q})} \bot$. Similarly, if $B \rightarrow l$ holds, then $B \wedge I^- \models_{\mathcal{LA}(\mathbb{Q})} \bot$, and so again $B \wedge I \models_{\mathcal{LA}(\mathbb{Q})} \bot$.

(iii) By the hypothesis, both I^+ and I^- contain only symbols common to A and B, so that $I \preceq A$ and $I \preceq B$.

□

Example 4.10. Consider the following sets of $\mathcal{LA}(\mathbb{Q})$ atoms:

$$A \stackrel{\text{def}}{=} \{(0 \neq x_1 - 3x_2 + 1), (0 = x_1 + x_2)\}$$
$$B \stackrel{\text{def}}{=} \{(0 = x_3 - 2x_1 - 1), (0 = 1 - 2x_3)\}.$$

In order to compute an interpolant for (A, B), we first split $n \stackrel{\text{def}}{=} (0 \neq x_1 - 3x_2 + 1)$ into $g \stackrel{\text{def}}{=} (0 < x_1 - 3x_2 + 1)$ and $l \stackrel{\text{def}}{=} (0 < -x_1 + 3x_2 - 1)$, thus obtaining A^+ and A^- defined as in Theorem 4.9. We then generate

two $\mathcal{LA}(\mathbb{Q})$-proofs of unsatisfiability P^+ for $A^+ \wedge B$ and P^- for $A^- \wedge B$, and replace g in P^+ with $g_\varepsilon \stackrel{\text{def}}{=} (0 \leq x_1 - 3x_2 + 1 - \varepsilon)$ and l in P^- with $l_\varepsilon \stackrel{\text{def}}{=} (0 \leq -x_1 + 3x_2 - 1 - \varepsilon)$, obtaining P^+_ε and P^-_ε:

$$P^+_\varepsilon \stackrel{\text{def}}{=} \dfrac{\dfrac{(0 \leq x_1 - 3x_2 + 1 - \varepsilon) \quad \dfrac{(0 = x_1 + x_2)}{(0 \leq x_1 + x_2)}}{(0 \leq 4x_1 + 1 - \varepsilon)} \quad \dfrac{\dfrac{(0 = x_3 - 2x_1 - 1)}{(0 \leq x_3 - 2x_1 - 1)} \quad \dfrac{(0 = 1 - 2x_3)}{(0 \leq 1 - 2x_3)}}{(0 \leq -4x_1 - 1)}}{(0 \leq -\varepsilon)}$$

$$P^-_\varepsilon \stackrel{\text{def}}{=} \dfrac{\dfrac{(0 \leq -x_1 + 3x_2 - 1 - \varepsilon) \quad \dfrac{(0 = x_1 + x_2)}{(0 \leq -x_1 - x_2)}}{(0 \leq -4x_1 - 1 - \varepsilon)} \quad \dfrac{\dfrac{(0 = x_3 - 2x_1 - 1)}{(0 \leq -x_3 + 2x_1 + 1)} \quad \dfrac{(0 = 1 - 2x_3)}{(0 \leq -1 + 2x_3)}}{(0 \leq +4x_1 + 1)}}{(0 \leq -\varepsilon)}$$

We then compute the two interpolants I^+_ε from P^+_ε and I^-_ε from P^-_ε:

$$I^+_\varepsilon \stackrel{\text{def}}{=} (0 \leq 4x_1 + 1 - \varepsilon) \qquad I^-_\varepsilon \stackrel{\text{def}}{=} (0 \leq -4x_1 - 1 - \varepsilon).$$

Therefore, according to Theorem 4.8 the two interpolants I^+ for (A^+, B) and I^- for (A^-, B) are:

$$I^+ \stackrel{\text{def}}{=} (0 < 4x_1 + 1) \qquad I^- \stackrel{\text{def}}{=} (0 < -4x_1 - 1).$$

Finally, since $n \in B$, according to Theorem 4.9, the interpolant I for (A, B) is

$$I \stackrel{\text{def}}{=} I^+ \vee I^- \equiv (0 < 4x_1 + 1) \vee (0 < -4x_1 - 1).$$

$\Diamond$

4.2.3 Obtaining stronger interpolants

We conclude this section by illustrating a simple technique for improving the strength of interpolants in $\mathcal{LA}(\mathbb{Q})$. The technique is orthogonal to our proof-generation algorithm described in §4.2.1, and it is therefore of

independent interest. It is an improvement of the general algorithm of [McM05] (and outlined in §4.1) for generating interpolants from $\mathcal{LA}(\mathbb{Q})$-proofs of unsatisfiability.

Definition 4.11. *Given two interpolants I_1 and I_2 for the same pair (A, B) of conjunctions of $\mathcal{LA}(\mathbb{Q})$-literals, we say that I_1 is* stronger *than I_2 if and only if $I_1 \models_{\mathcal{LA}(\mathbb{Q})} I_2$ but $I_2 \not\models_{\mathcal{LA}(\mathbb{Q})} I_1$.*

Our technique is based on the simple observation that the only purpose of the summations performed during the traversal of proof trees [5] for computing the interpolant (as described in §4.1) is that of eliminating A-local variables. In fact, it is easy to see that the conjunction of the constraints of A occurring as leaves in an $\mathcal{LA}(\mathbb{Q})$-proof of unsatisfiability satisfies the first two points of the definition of interpolant (Definition 4.1): if such constraints do not contain A-local variables, therefore, their conjunction is already an interpolant; if not, it suffices to perform only the summations of constraints of A that are necessary to eliminate A-local variables. Moreover, such interpolant is stronger than that obtained by performing all the summations in the proof tree, since for any set of constraints $\{s_1, \ldots, s_n\}$ and any set of positive coefficients $\{c_1, \ldots, c_n\}$, $s_1 \wedge \ldots \wedge s_n \models_{\mathcal{LA}(\mathbb{Q})} \sum_{i=1}^{n} c_i \cdot s_i$ holds.

According to this observation, our proposal can be described as: *perform only those summations which are necessary for eliminating A-local variables.*

Example 4.12. Consider the following sets of $\mathcal{LA}(\mathbb{Q})$-atoms:

$$A \stackrel{\text{def}}{=} \{(0 \leq x_1 - 3x_2 + 1), (0 \leq x_2 - \frac{1}{3}x_3), (0 \leq x_4 - \frac{3}{2}x_5 - 1)\}$$
$$B \stackrel{\text{def}}{=} \{(0 \leq 3x_5 - x_1), (0 \leq x_3 - 2x_4)\}$$

[5] corresponding to applications of the COMB rule.

and the following $\mathcal{LA}(\mathbb{Q})$-proof of unsatisfiability of $A \wedge B$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(0 \leq x_1 - 3x_2 + 1) \quad 3 \cdot (0 \leq x_2 - \frac{1}{3}x_3)}{(0 \leq x_1 - x_3 + 1)} \quad 2 \cdot (0 \leq x_4 - \frac{3}{2}x_5 - 1)}{(0 \leq x_1 - x_3 + 2x_4 - 3x_5 - 1)} \quad (0 \leq 3x_5 - x_1)}{(0 \leq -x_3 + 2x_4 - 1)} \quad (0 \leq x_3 - 2x_4)}{(0 \leq -1)}$$

Here, the variable x_2 is A-local, whereas all the others are AB-common. The interpolant computed with the algorithm of §4.1 is

$$(0 \leq x_1 - x_3 + 2x_4 - 3x_5 - 1),$$

which is the result of the linear combination of *all* the atoms of A in the proof. However, in order to eliminate the A-local variable x_2, it is enough to combine $(0 \leq x_1 - 3x_2 + 1)$ (with coefficient 1) and $(0 \leq x_2 - \frac{1}{3}x_3)$ (with coefficient 3), obtaining $(0 \leq x_1 - x_3 + 1)$. Therefore, a stronger interpolant is

$$(0 \leq x_1 - x_3 + 1) \wedge (0 \leq x_4 - \frac{3}{2}x_5 - 1).$$

$\Diamond$

The technique can be implemented with a small modification of the proof-based algorithm described in §4.1. We associate with each node in the proof P' (which is obtained from the original proof P by replacing inequalities from B with $(0 \leq 0)$) a list of pairs $\langle$*coefficient, inequality*$\rangle$. For a leaf, this list is a singleton in which the coefficient is 1 and the inequality is the atom in the leaf itself. For an inner node (which corresponds to an application of the COMB rule), the list l is generated from the two lists l_1 and l_2 of the premises as follows:

1. Set l as the concatenation of l_1 and l_2;
2. Let c_1 and c_2 be the coefficients used in the COMB rule. Multiply each coefficient c'_i occurring in a pair $\langle c'_i, 0 \leq t_i \rangle$ of l by c_1 if the pair comes from l_1, and by c_2 otherwise;

3. While there is an A-local variable x occurring in more than one pair $\langle c', 0 \leq t\rangle$ of l:[6]

   (a) Collect all the pairs $\langle c'_i, 0 \leq t_i\rangle$ in which x occurs;

   (b) Generate a new pair $p \stackrel{\text{def}}{=} \langle 1, 0 \leq \sum_i c'_i \cdot t_i\rangle$;

   (c) Add p to l, and remove all the pairs $\langle c'_i, 0 \leq t_i\rangle$.

After having applied the above algorithm, we can take the conjunction of the inequalities in the list associated with the root of P' as an interpolant.

Theorem 4.13. *Let P be a $\mathcal{LA}(\mathbb{Q})$-proof of unsatisfiability for a conjunction $A \wedge B$ of inequalities, and P' be obtained from P by replacing each inequality of B with $(0 \leq 0)$. Let $l \stackrel{def}{=} \langle c_1, 0 \leq t_1\rangle, \ldots, \langle c_n, 0 \leq t_n\rangle$ be the list associated with the root of P', computed as described above. Then $I \stackrel{def}{=} \bigwedge_{i=1}^{n}(0 \leq t_i)$ is an interpolant for (A, B). Moreover, I is always stronger than or equal to the interpolant obtained with the algorithm of §4.1 for the same proof P'.*

Proof. By induction on the structure of P', it is easy to prove that, for each constraint $(0 \leq t)$ in P' with its associated list $l \stackrel{\text{def}}{=} \langle c_1, 0 \leq t_1\rangle, \ldots, \langle c_n, 0 \leq t_n\rangle$:

1. $A \models \bigwedge_{i=1}^{n}(0 \leq t_i)$; and

2. $(0 \leq t) \equiv (0 \leq \sum_{i=1}^{n} c_i \cdot t_i)$

Since the root of P' is an interpolant for (A, B), this immediately proves the theorem. □

[6] That is, x occurs in t.

4.3 From SMT($\mathcal{DL}$) solving to SMT($\mathcal{DL}$) interpolation

Several interesting verification problems can be encoded using only the Difference Logic ($\mathcal{DL}$) subset of $\mathcal{LA}$, either over the rationals ($\mathcal{DL}(\mathbb{Q})$) or over the integers ($\mathcal{DL}(\mathbb{Z})$). As we have seen in §1.4.3, $\mathcal{DL}$ is much simpler than $\mathcal{LA}$, and many SMT solvers use dedicated, graph-based algorithms for checking the consistency of a set of $\mathcal{DL}(\mathbb{Q})$ atoms [CM06, NO05], by detecting the presence of negative-weight cycles in a graph-representation of the input set S of $\mathcal{DL}$ constraints, in which there is a vertex for each $\mathcal{DL}$ variable, and there exists an edge $x \xrightarrow{c} y$ for every $(0 \leq y - x + c) \in S$. [7]

In this section we present a specialized technique for computing interpolants in $\mathcal{DL}$ which exploits such state-of-the-art decision procedures, by extending the graph-based approach to generate interpolants. Since a set of weak inequalities in $\mathcal{DL}$ is consistent over the rationals if and only if it is consistent over the integers, our algorithm is applicable without any modifications to both $\mathcal{DL}(\mathbb{Q})$ and $\mathcal{DL}(\mathbb{Z})$ (see e.g. [NO05]).

Consider the interpolation problem (A, B) where A and B are sets of inequalities in the form $(0 \leq y - x + c)$, and let C be (the set of atoms in) a negative cycle in the graph corresponding to $A \cup B$.

If $C \subseteq A$, then A is inconsistent, in which case the interpolant is $\bot$. Similarly, when $C \subseteq B$, the interpolant is $\top$. If neither of these occurs, then the edges in the cycle can be partitioned in subsets of A and B. We call *maximal A-path* of C a path $x_1 \xrightarrow{c_1} \ldots \xrightarrow{c_{n-1}} x_n$ such that

(i) $x_i \xrightarrow{c_i} x_{i+1} \in A$ for $i \in [1, n-1]$, and

(ii) C contains $x' \xrightarrow{c'} x_1$ and $x_n \xrightarrow{c''} x''$ that are in B.

[7] Recall from §1.4.3 that we can assume w.l.o.g. that all constraints in S are in the form $(0 \leq y - x + c)$, where c is an integer constant.

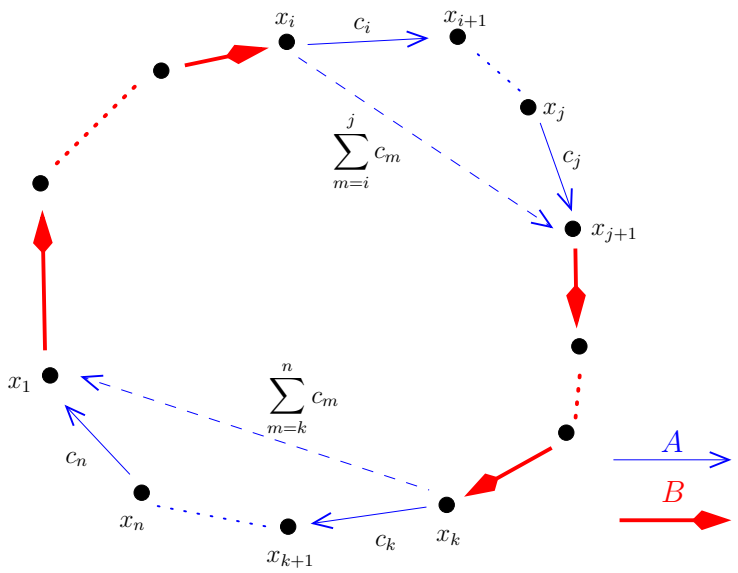


Figure 4.4: Generating a $\mathcal{DL}$-interpolant from a negative-weight cycle.

Clearly, the end-point variables x_1, x_n of the maximal A-path are such $x_1, x_n \preceq A$ and $x_1, x_n \preceq B$. Let the *summary constraint* of a maximal A-path $x_1 \xrightarrow{c_1} \ldots \xrightarrow{c_{n-1}} x_n$ be the inequality $0 \leq x_n - x_1 + \sum_{i=1}^{n-1} c_i$.

Theorem 4.14. *The conjunction of summary constraints of the A-paths of C is an interpolant for (A, B).*

Proof. Using the rules for $\mathcal{LA}(\mathbb{Q})$ of Figure 4.3, we build a deduction of the summary constraint of a maximal A-path from the conjunction of its corresponding set of constraints $\bigwedge_{i=1}^{n-1}(0 \leq x_{i+1} - x_i + c_i)$:

$$\dfrac{\dfrac{(0 \leq x_2 - x_1 + c_1) \quad (0 \leq x_3 - x_2 + c_2)}{(0 \leq x_3 - x_1 + c_1 + c_2)} \quad (0 \leq x_4 - x_3 + c_3)}{\dfrac{\ldots \qquad\qquad \ldots \qquad (0 \leq x_n - x_{n-1} + c_{n-1})}{(0 \leq x_n - x_1 + \sum_{i=1}^{n-1} c_i).}}$$

Hence, A entails the conjunction of the summary constraints of all maximal A-paths. Then, we notice that the conjunction of the summary constraints is inconsistent with B. In fact, the weight of a maximal A-path and the

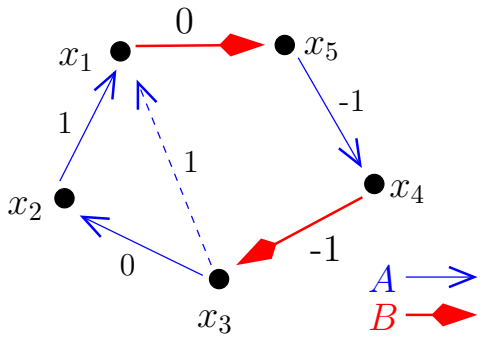
weight of its summary constraint are the same. Thus the cycle obtained from C by replacing each maximal A-path with the corresponding summary constraint is also a negative cycle. Finally, we notice that every variable x occurring in the conjunction of the summary constraints is an end-point variable, and thus $x \preceq A$ and $x \preceq B$. □

A final remark is in order. In principle, in order to generate a proof of unsatisfiability for a conjunction of $\mathcal{DL}(\mathbb{Q})$ atoms $A \wedge B$, the same rules used for $\mathcal{LA}(\mathbb{Q})$ [McM05] could be used. For instance, it is easy to build a proof which repeatedly applies the COMB rule with $c_1 = c_2 = 1$. In general, however, the interpolants generated from such proofs are not $\mathcal{DL}(\mathbb{Q})$ formulae anymore and, if computed starting from the same inconsistent set C, they are either identical or weaker than those generated with our method. In fact, it is easy to see that, unless our technique of §4.2.3 is adopted, such interpolants are in the form $(0 \leq \sum_i t_i)$ such that $\bigwedge_i (0 \leq t_i)$ is the corresponding interpolant generated with our graph-based method.

Example 4.15. Consider the following sets of $\mathcal{DL}(\mathbb{Q})$ atoms:

$$A \stackrel{\text{def}}{=} \{(0 \leq x_1 - x_2 + 1), (0 \leq x_2 - x_3), (0 \leq x_4 - x_5 - 1)\}$$

$$B \stackrel{\text{def}}{=} \{(0 \leq x_5 - x_1), (0 \leq x_3 - x_4 - 1)\}.$$



corresponding to the negative cycle on the right. It is straightforward to see from the graph that the resulting interpolant is $(0 \leq x_1 - x_3 + 1) \wedge (0 \leq x_4 - x_5 - 1)$, because the first conjunct is the summary constraint of the first two conjuncts in A.

Applying instead the rules of Figure 4.3 with coefficients 1, the proof of unsatisfiability is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(0 \leq x_1 - x_2 + 1) \quad (0 \leq x_2 - x_3)}{(0 \leq x_1 - x_3 + 1)} \qquad (0 \leq x_4 - x_5 - 1)}{(0 \leq x_1 - x_3 + x_4 - x_5)} \qquad (0 \leq x_5 - x_1)}{(0 \leq -x_3 + x_4)} \qquad (0 \leq x_3 - x_4 - 1)}{(0 \leq -1)}$$

By using the interpolation rules for $\mathcal{LA}(\mathbb{Q})$, the interpolant we obtain is $(0 \leq x_1 - x_3 + x_4 - x_5)$, which is not in $\mathcal{DL}(\mathbb{Q})$, and is weaker than that computed above:

$$\dfrac{\dfrac{\dfrac{\dfrac{(0 \leq x_1 - x_2 + 1) \quad (0 \leq x_2 - x_3)}{(0 \leq x_1 - x_3 + 1)} \qquad (0 \leq x_4 - x_5 - 1)}{(0 \leq x_1 - x_3 + x_4 - x_5)} \qquad (0 \leq 0)}{(0 \leq x_1 - x_3 + x_4 - x_5)} \qquad (0 \leq 0)}{(0 \leq x_1 - x_3 + x_4 - x_5)}$$

Notice that, if instead we apply our technique of §4.2.3, then the $\mathcal{LA}(\mathbb{Q})$-interpolant generated from the above proof is identical to the $\mathcal{DL}(\mathbb{Q})$ one above. $\Diamond$

4.4 From SMT($\mathcal{UTVPI}$) solving to SMT($\mathcal{UTVPI}$) interpolation

Another important subset of $\mathcal{LA}$ is given by the $\mathcal{UTVPI}$ theory (see §1.4.4), in which all constraints are in the form $(0 \leq \pm x_1 \pm x_2 + k)$, where k is an integer constant and variables x_i, x_2 range either over the rationals ($\mathcal{UTVPI}(\mathbb{Q})$) or over the integers ($\mathcal{UTVPI}(\mathbb{Z})$).

As for $\mathcal{DL}$, $\mathcal{UTVPI}$ can be treated more efficiently than the full $\mathcal{LA}$, and several specialized algorithms for $\mathcal{UTVPI}$ have been proposed in the

Table 4.1: The conversion map from $\mathcal{UTVPI}(\mathbb{Q})$ to $\mathcal{DL}(\mathbb{Q})$.

$\mathcal{UTVPI}(\mathbb{Q})$ **constraints**	$\mathcal{DL}(\mathbb{Q})$ **constraints**
$(0 \leq x_1 - x_2 + k)$	$(0 \leq x_1^+ - x_2^+ + k), (0 \leq x_2^- - x_1^- + k)$
$(0 \leq -x_1 - x_2 + k)$	$(0 \leq x_1^- - x_2^+ + k), (0 \leq x_2^- - x_1^+ + k)$
$(0 \leq x_1 + x_2 + k)$	$(0 \leq x_1^+ - x_2^- + k), (0 \leq x_2^+ - x_1^- + k)$
$(0 \leq -x_1 + k)$	$(0 \leq x_1^- - x_1^+ + 2 \cdot k)$
$(0 \leq x_1 + k)$	$(0 \leq x_1^+ - x_1^- + 2 \cdot k)$

literature. Traditional techniques are based on the iterative computation of the transitive closure of the constraints [HS97, JMSY94]; more recently [LM05] proposed a novel technique based on a reduction to $\mathcal{DL}$, so that graph-based techniques can be exploited, resulting into an asymptotically-faster algorithm. We adopt the latter approach and show how the graph-based interpolation technique of §4.3 can be extended to $\mathcal{UTVPI}$, for both the rationals (§4.4.1) and the integers (§4.4.2).

4.4.1 Graph-based interpolation for $\mathcal{UTVPI}$ on the Rationals

We analyze first the simpler case of $\mathcal{UTVPI}(\mathbb{Q})$. Miné [Min01] showed that it is possible to encode a set of $\mathcal{UTVPI}(\mathbb{Q})$ constraints into a $\mathcal{DL}(\mathbb{Q})$ one in a satisfiability-preserving way. The encoding works as follows. We use x_i to denote variables in the $\mathcal{UTVPI}(\mathbb{Q})$ domain and u, v for variables in the $\mathcal{DL}(\mathbb{Q})$ domain. For every variable x_i in $\mathcal{UTVPI}(\mathbb{Q})$, we introduce two distinct variables x_i^+ and x_i^- in $\mathcal{DL}(\mathbb{Q})$. We introduce a mapping Υ from $\mathcal{DL}(\mathbb{Q})$ variables to $\mathcal{UTVPI}(\mathbb{Q})$ signed variables, such that $\Upsilon(x_i^+) = x_i$ and $\Upsilon(x_i^-) = -x_i$. Υ extends to (sets of) constraints in the natural way: $\Upsilon(0 \leq ax_1 + bx_2 + k) \stackrel{\text{def}}{=} (0 \leq a\Upsilon(x_1) + b\Upsilon(x_2) + c)$, and $\Upsilon(\{c_i\}_i) \stackrel{\text{def}}{=} \{\Upsilon(c_i)\}_i$. We say that $(x_i^+)^- = x_i^-$ and $(x_i^-)^- = x_i^+$. We say that the constraints $(0 \leq u - v)$ and $(0 \leq (v)^- - (u)^-)$ such that $u, v \in \{x_i^+, x_i^-\}_i$ are *dual*. We encode each $\mathcal{UTVPI}$ constraint into the conjunction of two dual $\mathcal{DL}(\mathbb{Q})$ constraints,

as represented in Table 4.1. For each $\mathcal{DL}(\mathbb{Q})$ constraint $(0 \leq v - u + k)$, $(0 \leq \Upsilon(v) - \Upsilon(u) + k)$ is the corresponding $\mathcal{UTVPI}(\mathbb{Q})$ constraint. Notice that the two dual $\mathcal{DL}(\mathbb{Q})$ constraints in the right column of Table 4.1 are just different representations of the original $\mathcal{UTVPI}(\mathbb{Q})$ constraint. (The two dual constraints encoding a single-variable constraint are identical, so that their conjunction is collapsed into one constraint only.) The resulting set of constraints is satisfiable in $\mathcal{DL}(\mathbb{Q})$ if and only if the original one is satisfiable in $\mathcal{UTVPI}(\mathbb{Q})$ [Min01, LM05].

Consider the pair (A, B) where A and B are sets of $\mathcal{UTVPI}(\mathbb{Q})$ constraints. We apply the map of Table 4.1 and we encode (A, B) into a $\mathcal{DL}(\mathbb{Q})$ pair (A', B'), and build the constraint graph $G(A' \wedge B')$. If $G(A' \wedge B')$ has no negative cycle, we can conclude that $A' \wedge B'$ is $\mathcal{DL}(\mathbb{Q})$-consistent, and hence that $A \wedge B$ is $\mathcal{UTVPI}(\mathbb{Q})$-consistent; otherwise, $A' \wedge B'$ is $\mathcal{DL}(\mathbb{Q})$-inconsistent, and hence $A \wedge B$ is $\mathcal{UTVPI}(\mathbb{Q})$-inconsistent [Min01, LM05]. In fact, it is straightforward to observe that for any set of $\mathcal{DL}(\mathbb{Q})$ constraints $\{C_1, \ldots, C_n, C\}$ resulting from the encoding of some $\mathcal{UTVPI}(\mathbb{Q})$ constraints, if $\bigwedge_{i=1}^{n} C_i \models_{\mathcal{DL}(\mathbb{Q})} C$ then $\bigwedge_{i=1}^{n} \Upsilon(C_i) \models_{\mathcal{UTVPI}(\mathbb{Q})} \Upsilon(C)$.

When $A \wedge B$ is inconsistent, we can generate an $\mathcal{UTVPI}(\mathbb{Q})$-interpolant by extending the graph-based approach used for $\mathcal{DL}(\mathbb{Q})$.

Theorem 4.16. *Let $A \wedge B$ be an inconsistent conjunction of $\mathcal{UTVPI}(\mathbb{Q})$-constraints, and let $G(A' \wedge B')$ be the corresponding graph of $\mathcal{DL}(\mathbb{Q})$-constraints. Let I' be a $\mathcal{DL}(\mathbb{Q})$-interpolant built from $G(A' \wedge B')$ with the technique described in §4.3. Then $I \stackrel{def}{=} \Upsilon(I')$ is an interpolant for (A, B).*

Proof.

(i) I' is a conjunction of summary constraints, so it is in the form $\bigwedge_i C_i$. Therefore $A' \models_{\mathcal{DL}(\mathbb{Q})} C_i$ for all i, and so by the observation above $A \models_{\mathcal{UTVPI}(\mathbb{Q})} \Upsilon(C_i)$. Hence, $A \models_{\mathcal{UTVPI}(\mathbb{Q})} I$.

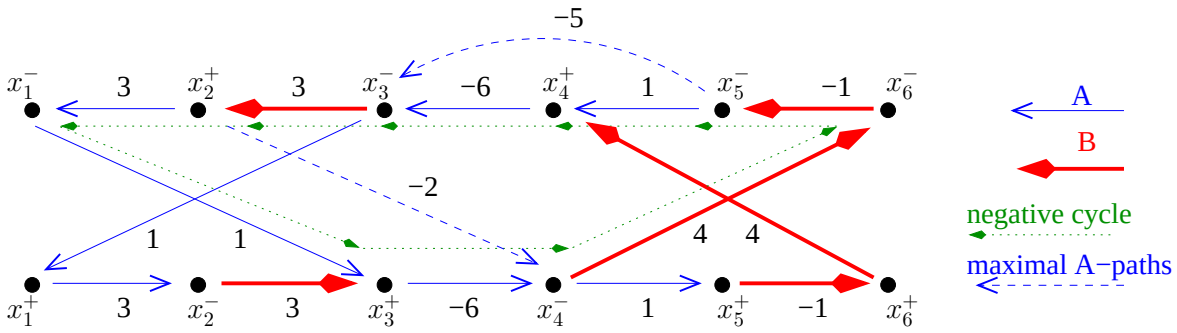


Figure 4.5: The constraint graph of Example 4.17. (We represent only one negative cycle with its corresponding A-paths, because the other is dual.)

(ii) From the $\mathcal{DL}(\mathbb{Q})$-inconsistency of $I' \wedge B'$ we immediately derive that $I \wedge B$ is $\mathcal{UTVPI}(\mathbb{Q})$-inconsistent.

(iii) $I \preceq A$ and $I \preceq B$ derive from $I' \preceq A'$ and $I' \preceq B'$ by the definitions of Υ and the map of Table 4.1.

□

As with the $\mathcal{DL}(\mathbb{Q})$ case, in principle, it is possible to generate a proof of unsatisfiability for a conjunction of $\mathcal{UTVPI}(\mathbb{Q})$ atoms $A \wedge B$ by repeatedly applying the Comb rule for $\mathcal{LA}(\mathbb{Q})$ [McM05] with $c_1 = c_2 = 1$. As with $\mathcal{DL}(\mathbb{Q})$, however, the interpolants generated from such proofs may not be $\mathcal{UTVPI}(\mathbb{Q})$ formulae anymore. Moreover, if computed starting from the same inconsistent set C and unless our technique of §4.2.3 is adopted, they are either identical or weaker than those generated with our graph-based method, since they are in the form $(0 \leq \sum_i t_i)$ such that $\bigwedge_i (0 \leq t_i)$ is the interpolant generated with our method.

Example 4.17. Consider the following sets of $\mathcal{UTVPI}(\mathbb{Q})$ constraints:

$$A = \{(0 \leq -x_2 - x_1 + 3), (0 \leq x_1 + x_3 + 1),$$
$$(0 \leq -x_3 - x_4 - 6), (0 \leq x_5 + x_4 + 1)\}$$

$$B = \{(0 \leq x_2 + x_3 + 3), (0 \leq x_6 - x_5 - 1), (0 \leq x_4 - x_6 + 4)\}$$

By the map of Table 4.1, they are converted into the following sets of $\mathcal{DL}(\mathbb{Q})$ constraints:

$$A' = \{(0 \leq x_1^- - x_2^+ + 3), (0 \leq x_2^- - x_1^+ + 3),$$
$$(0 \leq x_3^+ - x_1^- + 1), (0 \leq x_1^+ - x_3^- + 1),$$
$$(0 \leq x_4^- - x_3^+ - 6), (0 \leq x_3^- - x_4^+ - 6),$$
$$(0 \leq x_4^+ - x_5^- + 1), (0 \leq x_5^+ - x_4^- + 1)\}$$

$$B' = \{(0 \leq x_3^+ - x_2^- + 3), (0 \leq x_2^+ - x_3^- + 3),$$
$$(0 \leq x_6^+ - x_5^+ - 1), (0 \leq x_5^- - x_6^- - 1),$$
$$(0 \leq x_4^+ - x_6^+ + 4), (0 \leq x_6^- - x_4^- + 4)\}$$

whose conjunction corresponds to the constraint graph of Figure 4.5. This graph has a negative cycle

$$C' \stackrel{\text{def}}{=} x_2^+ \xrightarrow{3} x_1^- \xrightarrow{1} x_3^+ \xrightarrow{-6} x_4^- \xrightarrow{4} x_6^- \xrightarrow{-1} x_5^- \xrightarrow{1} x_4^+ \xrightarrow{-6} x_3^- \xrightarrow{3} x_2^+.$$

Thus, $A \wedge B$ is inconsistent in $\mathcal{UTVPI}(\mathbb{Q})$. From the negative cycle C' we can extract the set of A'-paths $\{x_2^+ \xrightarrow{-2} x_4^-,\ x_5^- \xrightarrow{-5} x_3^-\}$, corresponding to the formula $I' \stackrel{\text{def}}{=} (0 \leq x_4^- - x_2^+ - 2) \wedge (0 \leq x_3^- - x_5^- - 5)$, which is an interpolant for (A', B'). I' is thus mapped back into $I \stackrel{\text{def}}{=} \Upsilon(I') \stackrel{\text{def}}{=} (0 \leq -x_2 - x_4 - 2) \wedge (0 \leq x_5 - x_3 - 5)$, which is an interpolant for (A, B).

Applying instead the $\mathcal{LA}(\mathbb{Q})$ interpolation technique of [McM05], we find the interpolant $(0 \leq -x_2 - x_4 + x_5 - x_3 - 7)$, which is not in $\mathcal{UTVPI}(\mathbb{Q})$ and is strictly weaker than that computed with our method. $\Diamond$

4.4.2 Graph-based interpolation for $\mathcal{UTVPI}$ on the Integers

In order to deal with the more complex case of $\mathcal{UTVPI}(\mathbb{Z})$, we adopt a layered approach (see §1.3.1). First, we check the consistency in $\mathcal{UTVPI}(\mathbb{Q})$ using the technique of [Min01]. If this results in an inconsistency, we compute an $\mathcal{UTVPI}(\mathbb{Q})$-interpolant as described in §4.4.1. Clearly, this is also an interpolant in $\mathcal{UTVPI}(\mathbb{Z})$: condition (iii) is obvious, and conditions (i) and (ii) follow immediately from the fact that if an $\mathcal{UTVPI}$-formula is inconsistent over the rationals then it is inconsistent also over the integers. If the $\mathcal{UTVPI}(\mathbb{Q})$-procedure does not detect an inconsistency, we check the consistency in $\mathcal{UTVPI}(\mathbb{Z})$ using the algorithm proposed by Lahiri and Musuvathi in [LM05], which extends the ideas of [Min01] to the integer domain. In particular, it gives necessary and sufficient conditions to decide unsatisfiability by detecting particular kinds of zero-weight cycles in the induced $\mathcal{DL}$ constraint graph. This procedure works in $O(n \cdot m)$ time and $O(n + m)$ space, m and n being the number of constraints and variables respectively, which improves the previous $O(n^2 \cdot m)$ time and $O(n^2)$ space complexity of the previous procedure of [JMSY94].

We build on top of this algorithm and we extend the graph-based approach of §4.4.1 for producing interpolants also in $\mathcal{UTVPI}(\mathbb{Z})$. In particular, we use the following reformulation of a result of [LM05].

Theorem 4.18. *Let ϕ be a conjunction of $\mathcal{UTVPI}(\mathbb{Z})$ constraints such that ϕ is satisfiable in $\mathcal{UTVPI}(\mathbb{Q})$. Then ϕ is unsatisfiable in $\mathcal{UTVPI}(\mathbb{Z})$ if and only if the constraint graph $G(\phi)$ generated from ϕ has a cycle C of weight 0 containing two vertices x_i^+ and x_i^- such that the weight of the path $x_i^- \rightsquigarrow x_i^+$ along C is odd.*

Proof. The "only if" part is a corollary of lemmas 1, 2 and 4 in [LM05]. The "if" comes straightforwardly from the analysis done in [LM05], whose main intuitions we recall in what follows. Assume the constraint graph

$G(\phi)$ generated from ϕ has one cycle C of weight 0 containing two vertices x_i^+ and x_i^- such that the weight of the path $x_i^- \rightsquigarrow x_i^+$ along C is $2k+1$ for some integer value k. Since C has weight 0, the weight of the other path $x_i^+ \rightsquigarrow x_i^-$ along C is $-2k-1$. Then, the paths $x_i^- \rightsquigarrow x_i^+$ and $x_i^+ \rightsquigarrow x_i^-$ contain at least two constraints, because otherwise their weight would be even (see the last two rows of Table 4.1). Then, $x_i^- \rightsquigarrow x_i^+$ is in the form $x_i^- \rightsquigarrow v \xrightarrow{n} x_i^+$, for some v and n. From $x_i^- \rightsquigarrow v$, we can derive the summary constraint $(0 \leq v - x_i^- + (2k+1-n))$, which corresponds to the $\mathcal{UTVPI}(\mathbb{Z})$ constraint $(0 \leq \Upsilon(v) + x_i + (2k+1-n))$. (This corresponds to $l-2$ applications of the TRANSITIVE rule of [LM05], l being the number of constraints in $x_i^- \rightsquigarrow x_i^+$.) Then, by observing that the $\mathcal{UTVPI}(\mathbb{Z})$ constraint corresponding to $v \xrightarrow{n} x_i^+$ is $(0 \leq x_i - \Upsilon(v) + n)$, we can apply the TIGHTENING rule of [LM05] to obtain $(0 \leq x_i + \lfloor (2k+1-n+n)/2 \rfloor)$, which is equivalent to $(0 \leq x_i + k)$. Similarly, from $x_i^+ \rightsquigarrow x_i^-$ we can obtain $(0 \leq -x_i - k - 1)$, and thus an inconsistency using the CONTRADICTION rule of [LM05]. □

Consider a pair (A, B) of sets of $\mathcal{UTVPI}(\mathbb{Z})$-constraints such that $A \wedge B$ is consistent in $\mathcal{UTVPI}(\mathbb{Q})$ but inconsistent in $\mathcal{UTVPI}(\mathbb{Z})$. By Theorem 4.18, the constraint graph $G(A' \wedge B')$ has a cycle C of weight 0 containing two vertices x_i^+ and x_i^- such that the weight of the paths $x_i^- \rightsquigarrow x_i^+$ and $x_i^+ \rightsquigarrow x_i^-$ along C are $2k+1$ and $-2k-1$ respectively, for some value $k \in \mathbb{Z}$. Our algorithm computes an interpolant for (A, B) from the cycle C. Let C_A and C_B be the subsets of the edges in C corresponding to constraints in A' and B' respectively. We have to distinguish four distinct sub-cases.

Case 1: *x_i occurs in B but not in A.* Consequently, x_i^+ and x_i^- occur in B' but not in A', and hence they occur in C_B but not in C_A. Let I' be the conjunction of the summary constraints of the maximal C_A-paths, and let I be the conjunction of the corresponding $\mathcal{UTVPI}(\mathbb{Z})$ constraints. The

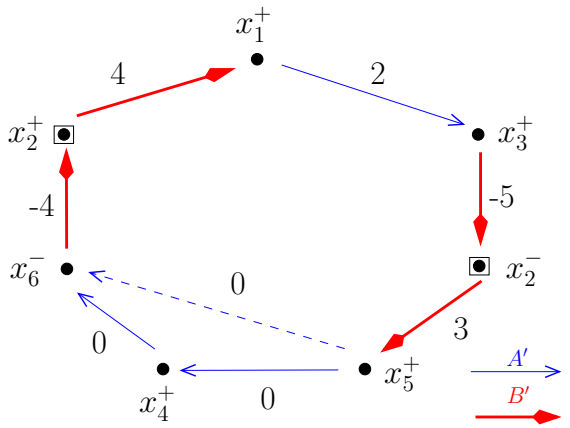


Figure 4.6: $\mathcal{UTVPI}(\mathbb{Z})$ interpolation, Case 1.

following theorem shows that I is an interpolant for (A, B).

Theorem 4.19. *Let (A, B) be a pair of sets of $\mathcal{UTVPI}(\mathbb{Z})$-constraints such that $A \wedge B \models_{\mathcal{UTVPI}(\mathbb{Z})} \bot$, let x_i be a variable that occurs in B but not in A, let $G(A' \wedge B')$ be the graph of the $\mathcal{DL}$-encoding of $A \wedge B$, and let C be a zero-weight cycle in the graph such that the weight of the paths $x_i^- \rightsquigarrow x_i^+$ and $x_i^+ \rightsquigarrow x_i^-$ along it are odd. Let I' be the conjunction of the summary constraints of the maximal C_A-paths, and let I be the conjunction of the corresponding $\mathcal{UTVPI}(\mathbb{Z})$-constraints. Then I is an interpolant for (A, B).*

Proof.

(i) By construction, $A \models_{\mathcal{UTVPI}(\mathbb{Z})} I$, as in §4.4.1.

(ii) The constraints in I' and C_B form a cycle matching the hypotheses of Theorem 4.18, from which $I \wedge B$ is $\mathcal{UTVPI}(\mathbb{Z})$-inconsistent.

(iii) We notice that every variable x_j^+, x_j^- occurring in the conjunction of the summary constraints is an end-point variable, so that $I' \preceq C_A$ and $I' \preceq C_B$, and thus $I \preceq A$ and $I \preceq B$. □

Example 4.20. Consider the following set of constraints:

$$S = \{(0 \leq x_1 - x_2 + 4), (0 \leq -x_2 - x_3 - 5), (0 \leq x_2 + x_6 - 4),$$
$$(0 \leq x_5 + x_2 + 3), (0 \leq -x_1 + x_3 + 2), (0 \leq -x_6 - x_4), (0 \leq x_4 - x_5)\},$$

partitioned into A and B as follows:

$$A = \begin{cases} (0 \leq x_3 - x_1 + 2) \\ (0 \leq -x_6 - x_4) \\ (0 \leq x_4 - x_5) \end{cases} \qquad B = \begin{cases} (0 \leq x_1 - x_2 + 4) \\ (0 \leq -x_2 - x_3 - 5) \\ (0 \leq x_2 + x_6 - 4) \\ (0 \leq x_5 + x_2 + 3) \end{cases}$$

Figure 4.6 shows a zero-weight cycle C in $G(A' \wedge B')$ such that the paths $x_2^- \rightsquigarrow x_2^+$ and $x_2^+ \rightsquigarrow x_2^-$ have an odd weight (-1 and 1 resp.) Therefore, by Theorem 4.18 $A \wedge B$ is $\mathcal{UTVPI}(\mathbb{Z})$-inconsistent. The two summary constraints of the maximal C_A paths are $(0 \leq x_6^- - x_5^+)$ and $(0 \leq x_3^+ - x_1^+ + 2)$. It is easy to see that $I = (0 \leq -x_6 - x_5) \wedge (0 \leq x_3 - x_1 + 2)$ is an $\mathcal{UTVPI}(\mathbb{Z})$-interpolant for (A, B). $\Diamond$

Case 2: *x_i occurs in both A and B.* Consequently, x_i^+ and x_i^- occur in both A' and B'. If neither x_i^+ nor x_i^- is such that both the incoming and outgoing edges belong to C_A, then the cycle obtained by replacing each maximal C_A-path with its summary constraint still contains both x_i^+ and x_i^-, so we can apply the same process of Case 1. Otherwise, if both the incoming and outgoing edges of x_i^+ belong to C_A, then we split the maximal C_A-path $u_1 \xrightarrow{c_1} \ldots \xrightarrow{c_k} x_i^+ \xrightarrow{c_{k+1}} \ldots \xrightarrow{c_n} u_n$ containing x_i^+ into the two parts which are separated by x_i^+: $u_1 \xrightarrow{c_1} \ldots \xrightarrow{c_k} x_i^+$ and $x_i^+ \xrightarrow{c_{k+1}} \ldots \xrightarrow{c_n} u_n$. We do the same for x_i^-. Let I' be the conjunction of the resulting summary constraints, and let I be corresponding set of $\mathcal{UTVPI}(\mathbb{Z})$ constraints. The following theorem shows that I is an interpolant for (A, B).

Theorem 4.21. *Let (A, B) be a pair of sets of $\mathcal{UTVPI}(\mathbb{Z})$-constraints such that $A \wedge B \models_{\mathcal{UTVPI}(\mathbb{Z})} \bot$, let x_i be a variable that occurs in both A and*

B, let $G(A' \wedge B')$ be the graph of the $\mathcal{DL}$-encoding of $A \wedge B$, and let C be a zero-weight cycle in the graph such that the weight of the paths $x_i^- \rightsquigarrow x_i^+$ and $x_i^+ \rightsquigarrow x_i^-$ along it are odd, and such that both the incoming and outgoing edges of x_i^+ in C belong to C_A. Let $u_1 \xrightarrow{c_1} \ldots \xrightarrow{c_k} x_i^+ \xrightarrow{c_{k+1}} \ldots \xrightarrow{c_n} u_n$ be the maximal C_A-patch containing x_i^+, and let s_1 and s_2 be respectively the summary constraints of the two parts of the above path which are separated by x_i^+. Let I' be the conjunction of s_1, s_2 and the summary constraints of the other maximal C_A-paths, and let I be the conjunction of the corresponding $\mathcal{UTVPI}(\mathbb{Z})$-constraints. Then I is an interpolant for (A, B).

Proof.

(i) As with Case 1, again, $A \models_{\mathcal{UTVPI}(\mathbb{Z})} I$.

(ii) Since we split the maximal C_A paths as described above, the constraints in I' and C_B form a cycle matching the hypotheses of Theorem 4.18, from which $I \wedge B$ is $\mathcal{UTVPI}(\mathbb{Z})$-inconsistent.

(iii) x_i^+, x_i^- occur in both A' and B' by hypothesis, and every other variable x_j^+, x_j^- occurring in the conjunction of the summary constraints is an end-point variable, so that $I' \preceq C_A$ and $I' \preceq C_B$, and thus $I \preceq A$ and $I \preceq B$. □

Example 4.22. Consider again the set of constraints S of Example 4.20, partitioned into A and B as follows:

$$A = \begin{cases} (0 \leq x_3 - x_1 + 2) \\ (0 \leq -x_6 - x_4) \\ (0 \leq x_2 + x_6 - 4) \\ (0 \leq x_1 - x_2 + 4) \end{cases} \qquad B = \begin{cases} (0 \leq -x_2 - x_3 - 5) \\ (0 \leq x_5 + x_2 + 3) \\ (0 \leq x_4 - x_5) \end{cases}$$

and the zero-weight cycle C of $G(A' \wedge B')$ shown in Figure 4.7. As in the previous example, there is a path $x_2^- \rightsquigarrow x_2^+$ of weight -1 and a path

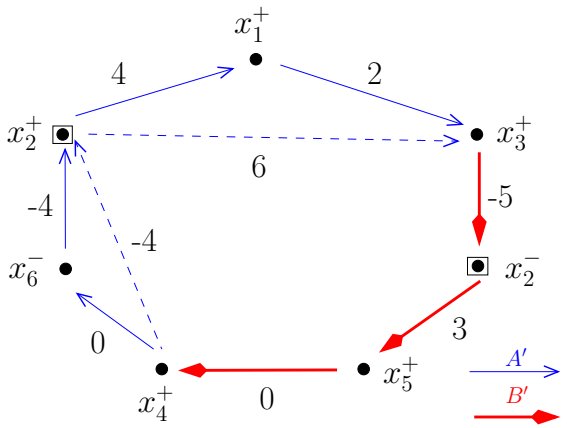


Figure 4.7: $\mathcal{UTVPI}(\mathbb{Z})$ interpolation, Case 2.

$x_2^+ \rightsquigarrow x_2^-$ of weight 1. In this case there is only one maximal C_A path, namely $x_4^+ \rightsquigarrow x_3^+$. Since the cycle obtained by replacing it with its summary constraint $(0 \leq x_3^+ - x_4^+ + 2)$ does not contain x_2^+, we split $x_4^+ \rightsquigarrow x_3^+$ into two paths, $x_4^+ \rightsquigarrow x_2^+$ and $x_2^+ \rightsquigarrow x_3^+$, whose summary constraints are $(0 \leq x_2^+ - x_4^+ - 4)$ and $(0 \leq x_3^+ - x_2^+ + 6)$ respectively. By replacing the two paths above with the two summary constraints, we get a zero-weight cycle which still contains the two odd paths $x_2^- \rightsquigarrow x_2^+$ and $x_2^+ \rightsquigarrow x_2^-$. Therefore, $I \stackrel{\text{def}}{=} (0 \leq x_2 - x_4 - 4) \wedge (0 \leq x_3 - x_2 + 6)$ is an interpolant for (A, B).

Notice that the $\mathcal{UTVPI}(\mathbb{Z})$-formula $J \stackrel{\text{def}}{=} (0 \leq x_3 - x_4 + 2)$ corresponding to the summary constraint of the maximal C_A path $x_4^+ \rightsquigarrow x_3^+$ is not an interpolant, since $J \wedge B$ is not $\mathcal{UTVPI}(\mathbb{Z})$-inconsistent. In fact, if we replace the maximal C_A path $x_4^+ \rightsquigarrow x_3^+$ with the summary constraint $x_4^+ \xrightarrow{2} x_3^+$, the cycle we obtain has still weight zero, but it contains no odd path between two variables x_i^+ and x_i^-. $\Diamond$

Case 3: *x_i occurs in A but not in B, and one of the paths $x_i^+ \rightsquigarrow x_i^-$ or $x_i^- \rightsquigarrow x_i^+$ in C contains only constraints of C_A.* In this case, x_i^+ and x_i^- occur in A' but not in B'. Suppose that $x_i^- \rightsquigarrow x_i^+$ consists only of constraints of C_A (the case $x_i^+ \rightsquigarrow x_i^-$ is analogous).

Let $2k+1$ be the weight of the path $x_i^- \rightsquigarrow x_i^+$ (which is odd by hypothesis), and let $\overline{C}$ be the cycle obtained by replacing such path with the edge $x_i^- \xrightarrow{2k} x_i^+$ in C. In the following, we call such a replacement *tightening summarization*. Since C has weight zero, $\overline{C}$ has negative weight. Let C^P be the set of $\mathcal{DL}$-constraints in the path $x_i^- \rightsquigarrow x_i^+$. Let I' be the $\mathcal{DL}$-interpolant computed from $\overline{C}$ for $(C_A \setminus C^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\}, C_B)$, and let I be the corresponding $\mathcal{UTVPI}(\mathbb{Z})$-formula. The following theorem shows that I is an interpolant for (A, B).

Theorem 4.23. *Let (A, B) be a pair of sets of $\mathcal{UTVPI}(\mathbb{Z})$-constraints such that $A \wedge B \models_{\mathcal{UTVPI}(\mathbb{Z})} \bot$, let x_i be a variable that occurs in A but not in B, let $G(A' \wedge B')$ be the graph of the $\mathcal{DL}$-encoding of $A \wedge B$, and let C be a zero-weight cycle in the graph such that the weight of the paths $x_i^- \rightsquigarrow x_i^+$ and $x_i^+ \rightsquigarrow x_i^-$ along it are odd, and such that all the constraints in the path $x_i^- \rightsquigarrow x_i^+$ belong to C_A. Let $\overline{C}$ be the cycle obtained by replacing the path $x_i^- \rightsquigarrow x_i^+$ with the edge $x_i^- \xrightarrow{2k} x_i^+$ in C, where $2k+1$ is the weight of the replaced path. Let C^P be the set of $\mathcal{DL}$-constraints in the path $x_i^- \rightsquigarrow x_i^+$. Let I' be the $\mathcal{DL}$-interpolant computed from $\overline{C}$ for $(C_A \setminus C^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\}, C_B)$, and let I be the corresponding $\mathcal{UTVPI}(\mathbb{Z})$-formula. Then I is an interpolant for (A, B).*

Proof.

(i) Let P be the set of $\mathcal{UTVPI}(\mathbb{Z})$ constraints in the path $x_i^- \rightsquigarrow x_i^+$. Since the weight $2k+1$ of such path is odd, we have that $P \models_{\mathcal{UTVPI}(\mathbb{Z})} (0 \leq x_i + k)$ (see page 136). Since $P \subseteq A$, therefore, $A \models_{\mathcal{UTVPI}(\mathbb{Z})} (0 \leq x_i + k)$. By observing that $(0 \leq x_i^+ - x_i^- + 2k)$ is the $\mathcal{DL}$-constraint corresponding to $(0 \leq x_i + k)$ we conclude that $C_A \setminus C^P \cup (0 \leq x_i^+ - x_i^- + 2k) \models_{\mathcal{DL}} I'$ implies that $A \setminus P \cup (0 \leq x_i + k) \models_{\mathcal{UTVPI}(\mathbb{Z})} I$, and so that $A \models_{\mathcal{UTVPI}(\mathbb{Z})} I$.

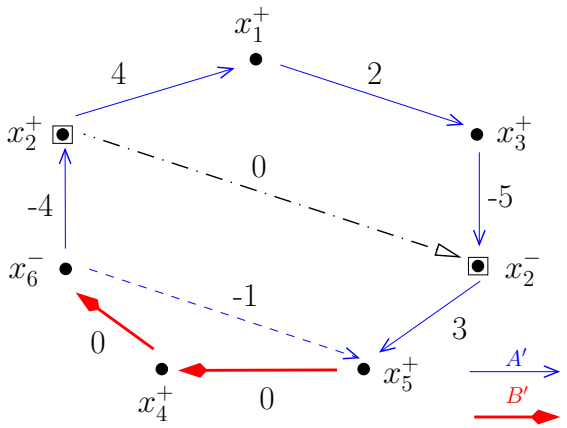


Figure 4.8: $\mathcal{UTVPI}(\mathbb{Z})$ interpolation, Case 3.

(ii) Since all the constraints in C_B occur in $\overline{C}$, we have that $B \wedge I$ is $\mathcal{UTVPI}(\mathbb{Z})$-inconsistent.

(iii) Since by hypothesis all the constraints in the path $x_i^- \rightsquigarrow x_i^+$ occur in C_A, from $I' \preceq (C_A \setminus C^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\})$ we have that $I \preceq A$. Finally, since all the constraints in C_B occur in $\overline{C}$, we have that $I \preceq B$. □

Example 4.24. Consider again the set S of constraints of Example 4.20, this time partitioned into A and B as follows:

$$A = \begin{cases} (0 \leq x_1 - x_2 + 4) \\ (0 \leq x_3 - x_1 + 2) \\ (0 \leq -x_2 - x_3 - 5) \\ (0 \leq x_2 + x_6 - 4) \\ (0 \leq x_5 + x_2 + 3) \end{cases} \qquad B = \begin{cases} (0 \leq -x_6 - x_4) \\ (0 \leq x_4 - x_5) \end{cases}$$

Figure 4.8 shows a zero-weight cycle C of $G(A' \wedge B')$. The only maximal C_A path is $x_6^- \rightsquigarrow x_5^+$. Since the path $x_2^+ \rightsquigarrow x_2^-$ has weight 1, we can add the tightening edge $x_2^+ \xrightarrow{1-1} x_2^-$ to $G(A' \wedge B')$ (shown in dots and dashes in Figure 4.8), corresponding to the constraint $(0 \leq x_2^- - x_2^+)$. Since all

constraints in the path $x_2^+ \rightsquigarrow x_2^-$ belong to A', $A' \models (0 \leq x_2^- - x_2^+)$. Moreover, the cycle obtained by replacing the path $x_2^+ \rightsquigarrow x_2^-$ with the tightening edge $x_2^+ \xrightarrow{0} x_2^-$ has a negative weight (-1). Therefore, we can generate a $\mathcal{DL}$-interpolant $I' \stackrel{\text{def}}{=} (0 \leq x_2^- - x_6^- - 4)$ from such cycle, which corresponds to the $\mathcal{UTVPI}(\mathbb{Z})$-interpolant $I \stackrel{\text{def}}{=} (0 \leq -x_2 + x_6 - 4)$.

Notice that, similarly to Example 4.22, also in this case we cannot obtain an interpolant from the summary constraint $(0 \leq x_5^+ - x_6^-)$ of the maximal C_A path $x_6^- \rightsquigarrow x_5^+$, as $(0 \leq x_5 + x_6) \wedge B$ is not $\mathcal{UTVPI}(\mathbb{Z})$-inconsistent. $\Diamond$

Case 4: *x_i occurs in A but not in B, and neither the path $x_i^+ \rightsquigarrow x_i^-$ nor the path $x_i^- \rightsquigarrow x_i^+$ in C consists only of constraints of C_A.* As in the previous case, x_i^+ and x_i^- occur in A' but not in B', and hence they occur in C_A but not in C_B. In this case, however, we can apply a tightening summarization neither to $x_i^+ \rightsquigarrow x_i^-$ nor to $x_i^- \rightsquigarrow x_i^+$, since none of the two paths consists only of constraints of C_A. We can, however, perform a *conditional tightening summarization* as follows. Let C_A^P and C_B^P be the sets of constraints of C_A and C_B respectively occurring in the path $x_i^- \rightsquigarrow x_i^+$, and let $\overline{C}_A^P$ and $\overline{C}_B^P$ be the sets of summary constraints of maximal paths in C_A^P and C_B^P. From $\overline{C}_A^P \cup \overline{C}_B^P$, we can derive $x_i^- \xrightarrow{2k} x_i^+$ (see Case 3), where $2k+1$ is the weight of the path $x_i^- \rightsquigarrow x_i^+$. Therefore, $\overline{C}_A^P \cup \overline{C}_B^P \models (0 \leq x_i^+ - x_i^- + 2k)$, and thus $\overline{C}_A^P \models \overline{C}_B^P \rightarrow (0 \leq x_i^+ - x_i^- + 2k)$. We say that $(0 \leq x_i^+ - x_i^- + 2k)$ is the summary constraint for $x_i^- \rightsquigarrow x_i^+$ *conditioned to* $\overline{C}_B^P$.

Using conditional tightening summarization, we generate an interpolant as follows. By replacing the path $x_i^- \rightsquigarrow x_i^+$ with $x_i^- \xrightarrow{2k} x_i^+$, we obtain a negative-weight cycle $\overline{C}$, as in Case 3. Let I' be the $\mathcal{DL}$-interpolant computed from $\overline{C}$ for $(C_A \setminus C_A^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\}, C_B \setminus C_B^P)$, and let I be the corresponding $\mathcal{UTVPI}(\mathbb{Z})$ formula. Finally, let $\overline{P}_B$ be the conjunction of $\mathcal{UTVPI}(\mathbb{Z})$ constraints corresponding to $\overline{C}_B^P$. The following

theorem show that $(\overline{P}_B \rightarrow I)$ is an interpolant for (A, B).

Theorem 4.25. *Let (A, B) be a pair of sets of $\mathcal{UTVPI}(\mathbb{Z})$-constraints such that $A \wedge B \models_{\mathcal{UTVPI}(\mathbb{Z})} \bot$, let x_i be a variable that occurs in A but not in B, let $G(A' \wedge B')$ be the graph of the $\mathcal{DL}$-encoding of $A \wedge B$, and let C be a zero-weight cycle in the graph such that the weight of the paths $x_i^- \rightsquigarrow x_i^+$ and $x_i^+ \rightsquigarrow x_i^-$ along it are odd, and such that none of such two paths consists only of constraints from C_A. Let C_A^P and C_B^P be the sets of constraints of C_A and C_B respectively occurring in the path $x_i^- \rightsquigarrow x_i^+$, and let $\overline{C}_A^P$ and $\overline{C}_B^P$ be the sets of summary constraints of maximal paths in C_A^P and C_B^P. Let $\overline{C}$ be the cycle obtained by replacing the path $x_i^- \rightsquigarrow x_i^+$ with the edge $x_i^- \xrightarrow{2k} x_i^+$ in C, where $2k+1$ is the weight of the replaced path. Let I' be the $\mathcal{DL}$-interpolant computed from $\overline{C}$ for $(C_A \setminus C_A^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\}, C_B \setminus C_B^P)$, and let I be the corresponding $\mathcal{UTVPI}(\mathbb{Z})$-formula. Let $\overline{P}_B$ be the conjunction of $\mathcal{UTVPI}(\mathbb{Z})$ constraints corresponding to $\overline{C}_B^P$. Then $(\overline{P}_B \rightarrow I)$ is an interpolant for (A, B).*

Proof.

(i) We know that $C_A \setminus C_A^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\} \models I'$, because I' is a $\mathcal{DL}$-interpolant. Moreover, $\overline{C}_A^P \cup \overline{C}_B^P \models (0 \leq x_i^+ - x_i^- + 2k)$, and so $C_A^P \cup \overline{C}_B^P \models (0 \leq x_i^+ - x_i^- + 2k)$. Therefore, $C_A \cup \overline{C}_B^P \models I'$, and thus $A \cup \overline{P}_B \models_{\mathcal{UTVPI}(\mathbb{Z})} I$, from which $A \models_{\mathcal{UTVPI}(\mathbb{Z})} (\overline{P}_B \rightarrow I)$.

(ii) Since I' is a $\mathcal{DL}$-interpolant for $(C_A \setminus C_A^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\}, C_B \setminus C_B^P)$, $I' \wedge (C_B \setminus C_B^P)$ is $\mathcal{DL}$-inconsistent, and thus $I \wedge B$ is $\mathcal{UTVPI}(\mathbb{Z})$-inconsistent. Since by construction $B \models_{\mathcal{UTVPI}(\mathbb{Z})} \overline{P}_B$, $(\overline{P}_B \rightarrow I) \wedge B$ is $\mathcal{UTVPI}(\mathbb{Z})$-inconsistent.

(iii) From $I' \preceq C_B \setminus C_B^P$ we have that $I \preceq B$, and from $I' \preceq C_A \setminus C_A^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\}$ that $I \preceq A$. Moreover, all the variables occurring in the constraints in $\overline{C}_B^P$ are end-point variables, so that $\overline{C}_B^P \preceq C_A$ and

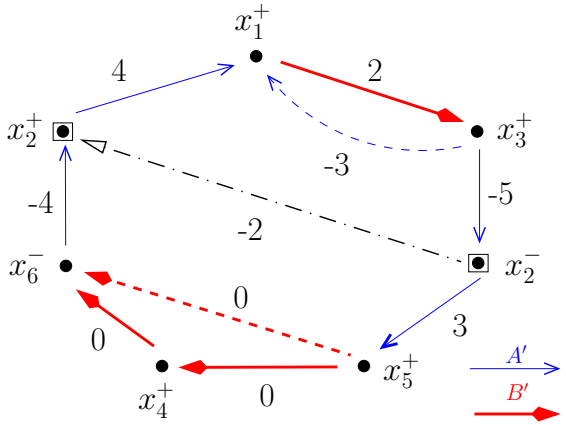


Figure 4.9: $\mathcal{UTVPI}(\mathbb{Z})$ interpolation, Case 4.

$\overline{C}_B^P \preceq C_B$, and thus $\overline{P}_B \preceq A$ and $\overline{P}_B \preceq B$. Therefore, $(\overline{P}_B \rightarrow I) \preceq A$ and $(\overline{P}_B \rightarrow I) \preceq B$. □

Example 4.26. We partition the set S of constraints of Example 4.20 into A and B as follows:

$$A = \begin{cases} (0 \leq x_1 - x_2 + 4) \\ (0 \leq -x_2 - x_3 - 5) \\ (0 \leq x_5 + x_2 + 3) \\ (0 \leq x_2 + x_6 - 4) \end{cases} \qquad B = \begin{cases} (0 \leq x_3 - x_1 + 2) \\ (0 \leq -x_6 - x_4) \\ (0 \leq x_4 - x_5) \end{cases}$$

Consider the zero-weight cycle C of $G(A' \wedge B')$ shown in Figure 4.9. In this case, neither the path $x_2^+ \rightsquigarrow x_2^-$ nor the path $x_2^- \rightsquigarrow x_2^+$ consists only of constraints of A', and thus we cannot use any of the two tightening edges $x_2^+ \xrightarrow{1-1} x_2^-$ and $x_2^- \xrightarrow{-1-1} x_2^+$ *directly* for computing an interpolant. However, we can compute the summary $x_2^- \xrightarrow{-2} x_2^+$ for $x_2^- \rightsquigarrow x_2^+$ *conditioned to* $x_5^+ \xrightarrow{0} x_6^-$, which is the summary constraint of the B-path $x_5^+ \rightsquigarrow x_6^-$, and whose corresponding $\mathcal{UTVPI}(\mathbb{Z})$ constraint is $(0 \leq -x_6 - x_5)$. By replacing the path $x_2^- \rightsquigarrow x_2^+$ with such summary, we obtain a negative-weight cycle $\overline{C}$, from which we generate the $\mathcal{DL}$-interpolant $(0 \leq x_1^+ - x_3^+ - 3)$,

corresponding to the $\mathcal{UTVPI}(\mathbb{Z})$ formula $(0 \leq x_1 - x_3 - 3)$. Therefore, the generated $\mathcal{UTVPI}(\mathbb{Z})$-interpolant is $(0 \leq -x_6 - x_5) \rightarrow (0 \leq x_1 - x_3 - 3)$.

As in Example 4.24, notice that we cannot generate an interpolant from the conjunction of summary constraints of maximal C_A paths, since the formula we obtain (i.e. $(0 \leq x_1 + x_6) \wedge (0 \leq x_5 - x_3 - 2)$) is not inconsistent with B. $\Diamond$

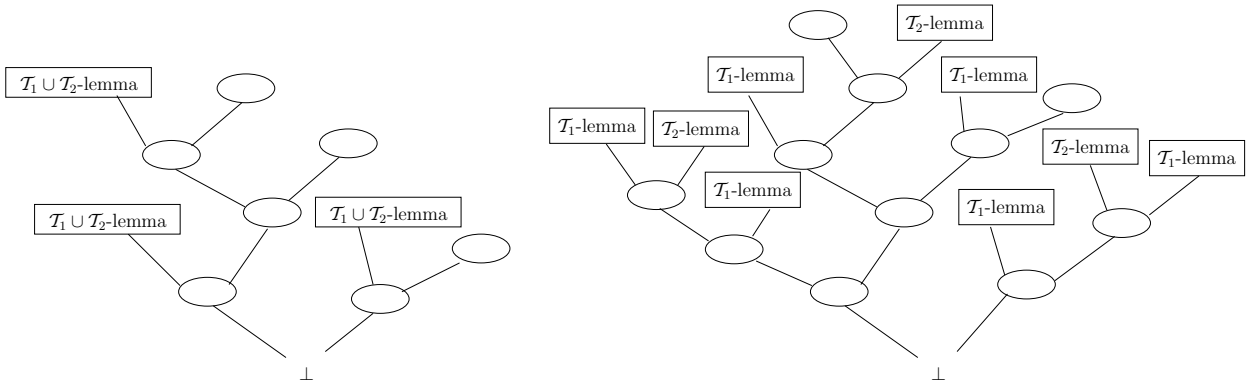
4.5 Computing interpolants for combined theories via DTC

In this section, we consider the problem of generating interpolants for a pair of $\mathcal{T}_1 \cup \mathcal{T}_2$-formulae (A, B), and propose a method based on the Delayed Theory Combination (DTC) approach [BBC$^+$06b, BCF$^+$08] (see §1.5.1). First, in §4.5.1 we provide some background on proof-generation in the N.O. and DTC combination methods, and recall from [YM05] the basics of interpolation for combined theories using N.O.; then, we present our novel technique for computing interpolants using DTC (§4.5.2); in §4.5.3 we discuss the advantages of the novel method; finally, in §4.5.4, we show how our novel technique can be used to generate multiple interpolants from the same proof.

4.5.1 Background

Resolution proofs with N.O. vs. resolution proofs with DTC

With an N.O.-based SMT solver (see §1.5.1), resolution proofs for formulae in a combination $\mathcal{T}_1 \cup \mathcal{T}_2$ of theories have the same structure as those for formulae in a single theory $\mathcal{T}$. The only difference is that theory lemmas in this case are the result of the N.O.-combination of $\mathcal{T}_1$ and $\mathcal{T}_2$ (i.e., they are $\mathcal{T}_1 \cup \mathcal{T}_2$-lemmas) (Figure 4.10 left). From the point of view of interpolation,



(NO) **(DTC)**

Figure 4.10: Different structures of resolution proofs of unsatisfiability for $\mathcal{T}_1 \cup \mathcal{T}_2$-formulae, using N.O. (left) and DTC (right).

the difference with respect to the case of a single theory $\mathcal{T}$ is that the $\mathcal{T}_1 \cup \mathcal{T}_2$-interpolants for the negations of the $\mathcal{T}_1 \cup \mathcal{T}_2$-lemmas can be computed with the combination method of [YM05] whenever it applies.

With DTC, resolution proofs are quite different from those obtained with N.O.. There is no $\mathcal{T}_1 \cup \mathcal{T}_2$-lemma anymore, because the two $\mathcal{T}_i$-solvers don't communicate directly. Instead, the proofs contain both $\mathcal{T}_1$-lemmas and $\mathcal{T}_2$-lemmas (Figure 4.10 right), and – importantly – *they contain also interface equalities*. (Notice that $\mathcal{T}_i$-lemmas derive either from $\mathcal{T}_i$-conflicts or from $\mathcal{T}_i$-propagation steps.) In this case, the combination of theories is encoded directly in the proofs (thanks to the presence of interface equalities), and not "hidden" in the $\mathcal{T}_1 \cup \mathcal{T}_2$-lemmas as with N.O.. This observation is at the heart of our DTC-based interpolant combination method.

Example 4.27. Consider the following formula ϕ:

$$\begin{aligned}\phi \stackrel{\text{def}}{=}\ & (a_1 = f(a_2)) \wedge (b_1 = f(b_2)) \wedge \\ & (y - a_2 = 1) \wedge (y - b_2 = 1) \wedge (a_1 + y = 0) \wedge (b_1 + y = 1).\end{aligned}$$

ϕ is expressed over the combined theory $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$: the first two atoms belong to $\mathcal{EUF}$, while the last four belong to $\mathcal{LA}(\mathbb{Q})$.

Using the N.O. combination method, ϕ can be proved unsatisfiable as follows:

1. From the conjunction $(y - a_2 = 1) \wedge (y - b_2 = 1)$, the $\mathcal{LA}(\mathbb{Q})$-solver deduces the interface equality $(a_2 = b_2)$, which is sent to the $\mathcal{EUF}$-solver;
2. From $(a_2 = b_2)$ and the conjunction $(a_1 = f(a_2)) \wedge (b_1 = f(b_2))$ the $\mathcal{EUF}$-solver deduces the interface equality $(a_1 = b_1)$, which is sent to the $\mathcal{LA}(\mathbb{Q})$-solver;
3. Together with the conjunction $(a_1 + y = 0) \wedge (b_1 + y = 1)$, $(a_1 = b_1)$ causes an inconsistency in the $\mathcal{LA}(\mathbb{Q})$-solver;
4. The $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ conflict-set generated is $\{(y - a_2 = 1), (y - b_2 = 1), (a_1 = f(a_2)), (b_1 = f(b_2)), (a_1 + y = 0), (b_1 + y = 1)\}$, corresponding to the $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$-lemma $C \stackrel{\text{def}}{=} \neg(y - a_2 = 1) \vee \neg(y - b_2 = 1) \vee \neg(a_1 = f(a_2)) \vee \neg(b_1 = f(b_2)) \vee \neg(a_1 + y = 0) \vee \neg(b_1 + y = 1)$.

The corresponding N.O. proof of unsatisfiability for ϕ is thus:

$$\dfrac{C \quad (b_1 + y = 1)}{\cdots \qquad (a_1 + y = 0)} \qquad \dfrac{\cdots \qquad (y - b_2 = 1)}{\cdots \qquad (y - a_2 = 1)} \qquad \dfrac{\cdots \qquad (b_1 = f(b_2))}{\cdots \qquad (a_1 = f(a_2))} \qquad \dfrac{}{\bot}$$

With DTC, the Boolean search space is augmented with the set of all possible interface equalities $Eq \stackrel{\text{def}}{=} \{(a_1 = a_2), (a_1 = b_1), (a_1 = b_2), (a_2 = b_1), (a_2 = b_2), (b_1 = b_2)\}$, so that the DPLL engine can branch on them. If we suppose that the negative branch is explored first (and we assume for simplicity that the $\mathcal{T}$-solvers do not perform deductions), using the DTC combination method ϕ can be proved unsatisfiable as follows:

1. Assigning $(a_2 = b_2)$ to false causes an inconsistency in the $\mathcal{LA}(\mathbb{Q})$-solver, which generates the $\mathcal{LA}(\mathbb{Q})$-lemma $C_1 \stackrel{\text{def}}{=} \neg(y - a_2 = 1) \vee \neg(y - b_2 = 1) \vee (a_2 = b_2)$. C_1 is used by the DPLL engine to backjump and unit-propagate $(a_2 = b_2)$;

2. After such propagation, assigning $(a_1 = b_1)$ to false causes an inconsistency in the $\mathcal{EUF}$-solver, which generates the $\mathcal{EUF}$-lemma $C_2 \stackrel{\text{def}}{=} \neg(a_1 = f(a_2)) \vee \neg(b_1 = f(b_2)) \vee \neg(a_2 = b_2) \vee (a_1 = b_1)$. C_2 is used by the DPLL engine to backjump and unit-propagate $(a_1 = b_1)$;

3. This propagation causes an inconsistency in the $\mathcal{LA}(\mathbb{Q})$-solver, which generates the $\mathcal{LA}(\mathbb{Q})$-lemma $C_3 \stackrel{\text{def}}{=} \neg(y - a_2 = 1) \vee \neg(y - b_2 = 1) \vee \neg(a_1 = b_1)$;

4. After learning C_3, the DPLL engine detects the unsatisfiability of ϕ.

The corresponding DTC proof of unsatisfiability for ϕ is thus:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{C_1 \quad (y - a_2 = 1)}{\cdots} \quad (y - b_2 = 1)}{\cdots} \quad C_2}{\cdots} \quad (b_1 = f(b_2))}{\cdots} \quad C_3}{\cdots} \quad (b_1 + y = 1)}{\cdots} \quad (a_1 + y = 0)}{\cdots} \quad (a_1 = f(a_2))}{\bot}$$

$\Diamond$

An important remark is in order. It is relatively easy to implement DTC in such a way that, if both $\mathcal{T}_1$ and $\mathcal{T}_2$ are convex, then all $\mathcal{T}$-lemmas generated contain at most one positive interface equality. This is due to the fact that for convex theories $\mathcal{T}$ it is possible to implement efficient $\mathcal{T}$-solvers which generate conflict sets containing at most one negated equality

between variables [BBC+05]. [8] (E.g., this is true for all the $\mathcal{T}_i$-solvers on convex theories implemented in MathSAT.) Thus, since we restrict to convex theories, in the rest of this section we can assume without loss of generality that every $\mathcal{T}$-lemma occurring as leaf in a resolution proof Π of unsatisfiability deriving from DTC contains at most one positive interface equality.

Interpolation with Nelson-Oppen

The work in [YM05] gives a method for generating an interpolant for a pair (A, B) of $\mathcal{T}_1 \cup \mathcal{T}_2$-formulae such that $A \wedge B \models_{\mathcal{T}_1 \cup \mathcal{T}_2} \bot$ by means of the N.O. schema. As in [YM05], we assume that A and B have been purified using disjoint sets of auxiliary variables. [9] We recall from [YM05] a couple of definitions.

Definition 4.28 (AB-mixed equality)**.** *An equality between variables $(a = b)$ is an AB-mixed equality if and only if $a \not\preceq B$ and $b \not\preceq A$ (or vice versa).*

Definition 4.29 (Equality-interpolating theory)**.** *A theory $\mathcal{T}$ is said to be equality-interpolating if and only if, for all A and B in $\mathcal{T}$ and for all AB-mixed equalities $(a = b)$ such that $A \wedge B \models_{\mathcal{T}} (a = b)$, there exists a term t such that $A \wedge B \models_{\mathcal{T}} (a = t) \wedge (t = b)$ and $t \preceq A$ and $t \preceq B$.*

The work in [YM05] describes procedures for computing the term t from an AB-mixed interface equality $(a = b)$ for some convex theories of interest, including $\mathcal{EUF}$, $\mathcal{LA}(\mathbb{Q})$, and the theory of lists.

Notationally, with the letters x, x_i, y, y_i, z we denote generic variables, whilst with the letters a, a_i, and b, b_i we denote variables such that $a_i \not\preceq B$ and $b_i \not\preceq A$; hence, with the letters e_i we denote generic AB-mixed interface

[8] We recall that, if $\mathcal{T}$ is convex, then $\mu \wedge \bigwedge_i \neg l_i \models_{\mathcal{T}} \bot$ if and only if $\mu \wedge \neg l_i \models_{\mathcal{T}} \bot$ for some i, where the l_i's are positive literals.

[9] We recall from §1.5.1 that purification is not strictly necessary.

equalities in the form $(a_i = b_i)$; with the letters η, η_i we denote conjunctions of literals where no AB-mixed interface equality occurs, and with the letters μ, μ_i we denote conjunctions of literals where AB-mixed interface equalities may occur. If μ_i (respectively η_i) is $\bigwedge_i l_i$, we write $\neg\mu_i$ (respectively $\neg\eta_i$) for the clause $\bigvee_i \neg l_i$.

Let $A \wedge B$ be a $\mathcal{T}_1 \cup \mathcal{T}_2$-inconsistent conjunction of $\mathcal{T}_1 \cup \mathcal{T}_2$-literals, such that $A \stackrel{\text{def}}{=} A_1 \wedge A_2$ and $B \stackrel{\text{def}}{=} B_1 \wedge B_2$ where each A_i and B_i is $\mathcal{T}_i$-pure. The N.O.-based method of [YM05] computes an interpolant for (A, B) by combining $\mathcal{T}_i$-specific interpolants for subsets of A, B and the set of entailed interface equalities $\{e_j\}_j$ that are exchanged between the $\mathcal{T}_i$-solvers for deciding the unsatisfiability of $A \wedge B$. In particular, let $Eq \stackrel{\text{def}}{=} \{e_j\}_j$ be the set of entailed interface equalities. Due to the fact that both $\mathcal{T}_1$ and $\mathcal{T}_2$ are equality-interpolating, it is possible to assume without loss of generality that Eq does not contain AB-mixed equalities, because instead of deducing an AB-mixed interface equality $(a = b)$, a $\mathcal{T}$-solver can always deduce the two corresponding equalities $(a = t) \wedge (t = b)$. (Notice that the other $\mathcal{T}$-solver treats the term t as if it were a variable [YM05].) Let $A' \stackrel{\text{def}}{=} A \cup (Eq \downarrow A)$ and $B' \stackrel{\text{def}}{=} B \cup (Eq \downarrow B)$. Then, $\mathcal{T}_i$-specific partial interpolants are combined according to the following inductive definition:

$$I_{A,B}(e) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } e \in A \\ \top & \text{if } e \in B \\ (I^i_{A',B'}(e) \vee \bigvee_{e_a \in A'} I_{A,B}(e_a)) \wedge \bigwedge_{e_b \in B'} I_{A,B}(e_b) & \text{otherwise,} \end{cases} \tag{4.9}$$

where e is either an entailed interface equality or $\bot$, and $I^i_{A',B'}(e)$ is a $\mathcal{T}_i$-interpolant for $(A' \cup \neg e, B')$ if $e \preceq A$, and for $(A', B' \cup \neg e)$ otherwise (if $e \preceq B$). The computed interpolant for (A, B) is then $I_{A,B}(\bot)$. We refer the reader to [YM05] for more details.

4.5.2 From DTC solving to DTC Interpolation

We now discuss how to extend the DTC method to interpolation. As with [YM05], we can handle the case that $\mathcal{T}_1$ and $\mathcal{T}_2$ are convex and equality-interpolating. The approach to generating interpolants for combined theories starts from the proof generated by DTC. Let Eq be the set of all interface equalities occurring in a DTC refutation proof for a $\mathcal{T}_1 \cup \mathcal{T}_2$-unsatisfiable formula $\phi \stackrel{\text{def}}{=} A \wedge B$.

In the case Eq does not contain AB-mixed equalities, that is, Eq can be partitioned into two sets $(Eq \setminus B) \stackrel{\text{def}}{=} \{(x = y) | (x = y) \preceq A$ and $(x = y) \not\preceq B\}$ and $(Eq \downarrow B) \stackrel{\text{def}}{=} \{(x = y) | (x = y) \preceq B\}$, *no interpolant-combination method is needed*: the combination is already encoded in the proof of unsatisfiability, and a direct application of Algorithm 4.1 to such proof yields an interpolant for the combined theory $\mathcal{T}_1 \cup \mathcal{T}_2$. Notice that this fact holds despite the fact that the interface equalities in Eq occur neither in A nor in B, but might be introduced in the resolution proof Π by $\mathcal{T}$-lemmas. In fact, as observed in [McM05], as long as for an atom p either $p \preceq A$ or $p \preceq B$ holds, it is possible to consider it part of A (respectively of B) simply by assuming the tautology clause $p \vee \neg p$ to be part of A (respectively of B). Therefore, we can treat the interface equalities in $(Eq \setminus B)$ as if they appeared in A, and those in $(Eq \downarrow B)$ as if they appeared in B.

When Eq contains AB-mixed equalities, instead, a proof-rewriting step is performed in order to obtain a proof which is free from AB-mixed equalities, that is amenable for interpolation as described above. The idea is similar to that used in [YM05] in the case of N.O.: using the fact that $\mathcal{T}_1$ and T_2 are equality-interpolating, we reduce this case to the previous one by "splitting" every AB-mixed interface equality $(a_i = b_i)$ into the conjunction of two parts $(a_i = t_i) \wedge (t_i = b_i)$, such that $(a_i = t_i) \preceq A$ and

$(t_i = b_i) \preceq B$. The main difference is that we do this *a posteriori*, after the construction of the resolution proof of unsatisfiability Π. In order to do this, we traverse Π and split each AB-mixed equality, performing also the necessary manipulations to ensure that the result is still a resolution proof of unsatisfiability.

We describe this process in two steps. In §4.5.2 we introduce a particular kind of resolution proofs of unsatisfiability, called ie-local, and show how to eliminate AB-mixed interface equalities from ie-local proofs; in §4.5.2 we show how to implement a variant of DTC so that to generate ie-local proofs.

Eliminating AB-mixed equalities by exploiting ie-locality

Definition 4.30 (ie-local proof)**.** *A resolution proof of unsatisfiability* Π *is local with respect to interface equalities (*ie*-local) if and only if the interface equalities occur only in subproofs* Π_i^{ie} *of* Π*, such that within each* Π_i^{ie}*:*

(i) all leaves are also $\mathcal{T}$*-lemma leaves of* Π*;*

(ii) all the pivots are interface equalities;

(iii) the root contains no interface equality;

(iv) every right premise of an inner node is a leaf $\mathcal{T}$*-lemma containing exactly one positive interface equality.* [10]

As a consequence of this definition, we also have that, within each Π_i^{ie} in Π:

(v) all nodes are $\mathcal{T}_1 \cup \mathcal{T}_2$-valid; (*Proof sketch*: they result from Boolean resolution steps from $\mathcal{T}_1$-valid and $\mathcal{T}_2$-valid clauses, hence they are $\mathcal{T}_1 \cup \mathcal{T}_2$-valid.)

[10] We have adopted the graphical convention that at each resolution step in a Π_i^{ie} subproof, if $(a_i = b_i)$ is the pivot, then the premises containing $\neg(a_i = b_i)$ and $(a_i = b_i)$ are the left and right premises respectively.

(vi) the only leaf $\mathcal{T}$-lemma which is a left premise contains no positive interface equality. (*Proof sketch:* we notice that, in a resolution step $\frac{C_1 \; C_2}{C_3}$, if C_3 contains no positive interface equality, at least one between C_1 and C_2 contains no positive interface equality; since by (iv) the right premise contains one positive interface equality, only the left premise contains no positive interface equality. Thus the leftmost leaf $\mathcal{T}$-lemma of Π_i^{ie} contains no positive interface equality.)

(vii) if an interface equality e_j occurs negatively in some $\mathcal{T}$-lemma C_j, then e_j occurs positively in a leaf $\mathcal{T}$-lemma C_k which is the right premise of a resolution step whose left premise derives from C_j and other $\mathcal{T}$-lemmas. (*Proof sketch:* suppose that $\neg e_j$ occurs in C_j but e_j does not occur in any such C_k. Then e_j can not be a pivot, hence $\neg e_j$ occurs in the root of Π_i^{ie}, thus violating (iii).)

Intuitively, in ie-local proofs of unsatisfiability all the reasoning on interface equalities is circumscribed within Π_i^{ie} subproofs, which are linear sub-proofs involving only $\mathcal{T}$-lemmas as leaves, starting from the one containing no positive interface equality, each time eliminating one negative interface equality by resolving it against the only positive one occurring in another leaf $\mathcal{T}$-lemma.

Example 4.31. Consider the $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ formula ϕ of Example 4.27, and the $\mathcal{T}$-lemmas C_1, C_2 and C_3 introduced by DTC to prove its unsatisfiability. The proof Π of Example 4.27 is not ie-local, because resolution steps involving interface equalities are interleaved with resolution steps involving other atoms. The following proof Π', instead, is ie-local: all the interface equalities are used as pivots in the Π^{ie} subproof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{C_3 \quad C_2}{\ldots}\,[\text{pivot } (a_1 = b_1)] \quad C_1}{\ldots}\,[\text{pivot } (a_2 = b_2)]}{\ldots}\,\Pi^{\mathsf{ie}} \quad (a_2 + z = 1)}{\ldots \qquad (a_1 + z = 0)}}{\ldots \qquad (z - x_2 = 1)}}{\ldots \qquad (a_1 = f(x_1))}}{\ldots \qquad (a_2 = f(x_2))}}{\ldots \qquad (z - x_1 = 1)}}{\bot}$$

$C_1 \stackrel{\text{def}}{=} (\underline{a_2 = b_2}) \vee \neg(y - a_2 = 1) \vee \neg(y - b_2 = 1)$
$C_2 \stackrel{\text{def}}{=} (\underline{a_1 = b_1}) \vee \neg(b_1 = f(b_2)) \vee \neg(a_1 = f(a_2)) \vee \neg(\underline{a_2 = b_2})$
$C_3 \stackrel{\text{def}}{=} \neg(a_1 + y = 0) \vee \neg(b_1 + y = 1) \vee \neg(\underline{a_1 = b_1})$.

$\Diamond$

If Π is an ie-local proof containing AB-mixed interface equalities, then it is possible to eliminate all of them from Π by applying Algorithm 4.11 to every Π^{ie}_i subproof of Π. In a nutshell, each Π^{ie}_i subproof is explored bottom-up, starting from the right premise of the root, each time expanding the rightmost side $\mathcal{T}$-lemma in the form $C_i \stackrel{\text{def}}{=} (a_i = b_i) \vee \neg\eta_i$, such that $(a_i = b_i)$ is AB-mixed, into the (implicit) conjunction of two novel $\mathcal{T}$-lemmas $C'_i \stackrel{\text{def}}{=} (a_i = t_i) \vee \neg\eta_i$ and $C''_i \stackrel{\text{def}}{=} (t_i = b_i) \vee \neg\eta_i$ (step (4)), where t_i is the AB-pure term computed from C_i as described in §4.5.1. Then the resolution step against C_i is substituted with the concatenation of two resolution steps against C'_i and C''_i (step (5)) and then the substitution $\neg(a_i = b_i) \longmapsto \neg(a_i = t_i) \vee \neg(t_i = b_i)$ is propagated bottom-up along the left subproof Π. Notice that C'_i and C''_i are still $\mathcal{T}_i$-valid because $\mathcal{T}_i$ is Equality-interpolating and η_i does not contain other AB-mixed interfaced equalities.

Example 4.32. Consider the formula ϕ of Example 4.27 and its ie-local proof of unsatisfiability of Example 4.31. Suppose that ϕ is partitioned as

Algorithm 4.11: Rewriting of Π^{ie} subproofs

1. Let σ be a mapping from negative AB-mixed interface equalities to a disjunction of two negative interface equalities, such that $\sigma[\neg(a_i = b_i)] \mapsto \neg(a_i = t_i) \vee \neg(t_i = b_i)$ and t_i is an AB-pure term as described in §4.5.1. Initially, σ is empty.

2. Let $C_i \stackrel{\text{def}}{=} (a_i = b_i) \vee \neg\mu_i$ be the right premise $\mathcal{T}$-lemma of the root of the Π^{ie} subproof.

3. Replace each $\neg(a_j = b_j)$ in C_i with $\sigma[\neg(a_j = b_j)]$, to obtain $C_i^* \stackrel{\text{def}}{=} (a_i = b_i) \vee \neg\eta_i$. If $(a_i = b_i)$ is not AB-mixed, then let Π be the subproof rooted in the left premise, and go to step (7).

4. Split C_i^* into $C_i' \stackrel{\text{def}}{=} (a_i = t_i) \vee \neg\eta_i$ and $C_i'' \stackrel{\text{def}}{=} (t_i = b_i) \vee \neg\eta_i$.

5. Rewrite the subproof

$$\dfrac{\dfrac{\vdots}{\neg(a_i = b_i) \vee \neg\mu_k} \quad C_i}{\neg\mu_k \vee \neg\mu_i} \quad \text{into} \quad \dfrac{\dfrac{\boxed{\dfrac{\vdots}{\neg(a_i = t_i) \vee \neg(t_i = b_i) \vee \neg\mu_k}\,\Pi} \quad C_i'}{\neg(t_i = b_i) \vee \neg\eta_k \vee \neg\eta_i} \quad C_i''}{\neg\eta_k \vee \neg\eta_i}$$

   where $\neg\eta_k$ is obtained by $\neg\mu_k$ by substituting each negative AB-mixed interface equality $\neg(a_j = b_j)$ with $\sigma[\neg(a_j = b_j)]$.

6. Update σ by setting $\sigma[\neg(a_i = b_i)]$ to $\neg(a_i = t_i) \vee \neg(t_i = b_i)$.

7. If Π is of the form $\dfrac{\vdots \; C_j}{\cdots}$, set C_i to C_j and go to step (3).

8. Otherwise, Π is the leaf $\neg(a_i = t_i) \vee \neg(t_i = b_i) \vee \neg\mu_k$. In this case, replace each $\neg(a_j = b_j)$ in $\neg\mu_k$ with $\sigma[\neg(a_j = b_j)]$, and then exit.

follows:

$$\begin{aligned}
\phi &\stackrel{\text{def}}{=} A \wedge B \\
A &\stackrel{\text{def}}{=} (a_1 = f(a_2)) \wedge (y - a_2 = 1) \wedge (a_1 + y = 0) \\
B &\stackrel{\text{def}}{=} (b_1 = f(b_2)) \wedge (y - b_2 = 1) \wedge (b_1 + y = 1)
\end{aligned}$$

In this case, both interface equalities $(a_1 = b_1)$ and $(a_2 = b_2)$ are AB-mixed. Consider the Π^{ie} subproof of Example 4.31:

$$C_1 \stackrel{\text{def}}{=} (\underline{a_2 = b_2}) \vee \neg(y - a_2 = 1) \vee \neg(y - b_2 = 1)$$
$$C_2 \stackrel{\text{def}}{=} (\underline{a_1 = b_1}) \vee \neg(b_1 = f(b_2)) \vee \neg(a_1 = f(a_2)) \vee \neg(\underline{a_2 = b_2})$$
$$C_3 \stackrel{\text{def}}{=} \neg(a_1 + y = 0) \vee \neg(b_1 + y = 1) \vee \neg(\underline{a_1 = b_1})$$

$$\dfrac{\dfrac{C_3 \quad C_2}{\Theta_1}\quad C_1}{\Theta_2}\ \Pi^{\mathsf{ie}}$$

$$\Theta_1 \stackrel{\text{def}}{=} \neg(a_1 + y = 0) \vee \neg(b_1 + y = 1) \vee \neg(b_1 = f(b_2)) \vee \neg(a_1 = f(a_2)) \vee \neg(a_2 = b_2)$$
$$\Theta_2 \stackrel{\text{def}}{=} \neg(a_1 + y = 0) \vee \neg(b_1 + y = 1) \vee \neg(b_1 = f(b_2)) \vee \neg(a_1 = f(a_2)) \vee \neg(y - a_2 = 1) \vee \neg(y - b_2 = 1).$$

The first $\mathcal{T}$-lemma processed by Algorithm 4.11 is C_1. Using the technique of [YM05], $(a_2 = b_2)$ is split into $(a_2 = y - 1) \wedge (y - 1 = b_2)$ (step (4)), thus obtaining C_1', C_1'' and the new proof (in step (5)):

$$C_1' \stackrel{\text{def}}{=} (a_2 = y - 1) \vee \neg(y - a_2 = 1) \vee \neg(y - b_2 = 1)$$
$$C_1'' \stackrel{\text{def}}{=} (y - 1 = b_2) \vee \neg(y - a_2 = 1) \vee \neg(y - b_2 = 1)$$
$$\Theta_2' \stackrel{\text{def}}{=} \neg(y - 1 = b_2) \vee \neg(a_1 + y = 0) \vee \neg(b_1 + y = 1) \vee \neg(b_1 = f(b_2)) \vee \neg(a_1 = f(a_2)) \vee \neg(y - a_2 = 1) \vee \neg(y - b_2 = 1).$$

$$\dfrac{\dfrac{\dfrac{C_3 \quad C_2}{\Theta_1}\quad C_1'}{\Theta_2'}\quad C_1''}{\Theta_2}$$

Then, $\sigma[\neg(a_2 = b_2)]$ is set to $\neg(a_2 = y - 1) \vee \neg(y - 1 = b_2)$ (step (6)), and a new iteration of the loop (3)-(7) is performed, this time processing C_2. First, $\neg(a_2 = b_2)$ is replaced by $\neg(a_2 = y - 1) \vee \neg(y - 1 = b_2)$ (step (3)). Then, $(a_1 = b_1)$ can be split into $(a_1 = f(y - 1)) \wedge (f(y - 1) = b_1)$ (step (4)). After the rewriting of step (5), the proof is:

$$C_2' \stackrel{\text{def}}{=} (a_1 = f(y - 1)) \vee \neg(b_1 = f(b_2)) \vee \neg(a_1 = f(a_2)) \vee \neg(a_2 = y - 1) \vee \neg(y - 1 = b_2)$$
$$C_2'' \stackrel{\text{def}}{=} (f(y - 1) = b_1) \vee \neg(b_1 = f(b_2)) \vee \neg(a_1 = f(a_2)) \vee \neg(a_2 = y - 1) \vee \neg(y - 1 = b_2)$$
$$\Theta_1' \stackrel{\text{def}}{=} \neg(a_1 + y = 0) \vee \neg(b_1 + y = 1) \vee \neg(b_1 = f(b_2)) \vee \neg(a_1 = f(a_2)) \vee \neg(a_2 = y - 1) \vee \neg(y - 1 = b_2)$$
$$\Theta_1'' \stackrel{\text{def}}{=} \neg(a_1 = f(y - 1)) \vee \neg(a_1 + y = 0) \vee \neg(b_1 + y = 1) \vee \neg(b_1 = f(b_2)) \vee \neg(a_1 = f(a_2)) \vee \neg(a_2 = b_2)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{C_3 \quad C_2'}{\Theta_1''}\quad C_2''}{\Theta_1'}\quad C_1'}{\Theta_2'}\quad C_1''}{\Theta_2}$$

Finally, C_3 is processed in step (8), $\neg(a_1 = b_1)$ is replaced with $\neg(a_1 =$

$f(y-1)) \vee \neg(f(y-1) = b_1)$, and the following final proof Π'^{ie} is generated:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{C'_3 \quad C'_2}{\Theta''_1} \quad C''_2}{\Theta'_1} \quad C'_1}{\Theta'_2} \quad C''_1}{\Theta_2}$$

such that $C'_3 \stackrel{\text{def}}{=} C_3[\neg(a_1 = b_1) \mapsto \neg(a_1 = f(y-1)) \vee \neg(f(y-1) = b_1)]$. $\Diamond$

The following theorem states that Algorithm 4.11 is correct.

Theorem 4.33. *Let* Π *be a* Π^{ie} *subproof, and let* Π' *be the result of applying Algorithm 4.11 to* Π. *Then:*

(a) Π' *does not contain any* AB*-mixed interface equality; and*

(b) Π' *is a valid subproof with the same root as* Π.

Proof.

(a) Consider the $\mathcal{T}$-lemma C_i of Step (3). By item (vii) of Definition 4.30, all negative interface equalities occurring in C_i occur positively in leaf $\mathcal{T}$-lemmas that are closer to the root of Π. For the same reason, the first $\mathcal{T}$-lemma C_i analyzed in step (2) contains no negative AB-mixed interface equalities. Therefore, it follows by induction that all negative AB-mixed interface equalities in C_i must have been split in Step (4) of a previous iteration of the loop (3)-(7) of Algorithm 4.11, and thus they occur in σ. The same argument can be used to show also that at steps (5) and (8) every negative AB-mixed interface equality in $\neg\mu_k$ occurs in σ.

(b) We show that:

(i) Every substep $\dfrac{\Theta' \quad \Theta''}{\Theta'''}$ of Π' is a valid resolution step;

(ii) every leaf of Π' is a $\mathcal{T}$-lemma; and

(iii) the root of Π' is the same as that of Π.

(i) The only problematic case is the resolution step

$$\frac{\neg(a_i = t_i) \vee \neg(t_i = b_i) \vee \neg\mu_k \quad C_i'}{\neg(t_i = b_i) \vee \neg\eta_k \vee \neg\eta_i}$$

introduced in step (5) of Algorithm 4.11. In this case, we have to show that at the end of the algorithm, all the negative AB-mixed interface equalities in $\neg\mu_k$ have been replaced such that the result is identical to $\neg\eta_k$. We already know that all negative AB-mixed equalities in $\neg\mu_k$ occur in σ, thus we only have to show that $\sigma[\neg e_j]$ cannot change between the time when $\neg e_j$ was rewritten to obtain $\neg\eta_k$ and the time in which it is rewritten in $\neg\mu_k$. The negative equality $\neg e_j$ is replaced in $\neg\mu_k$ at the next iteration of the algorithm (in step (5) for inner nodes, and in step (8) for the final leaf). In the meantime, the only update to σ is performed in step (6), but it involves the negative equality $\neg(a_i = b_i)$, which does not occur in $\neg\mu_k$.

(ii) Let C_i be a $\mathcal{T}$-lemma in Π. First, we observe that if $C_i \equiv \neg(a_i = b_i) \vee \neg\mu_i$, then for any t_i also the clause $C_i^* \stackrel{\text{def}}{=} \neg(a_i = t_i) \vee \neg(t_i = b_i) \vee \neg\mu_i$ is a $\mathcal{T}$-lemma, since $(a_i = t_i) \wedge (t_i = b_i) \models_{\mathcal{T}} (a_i = b_i)$ by transitivity. Therefore, it follows by induction on the number of substitutions that the clauses obtained in steps (3) and (8) of Algorithm 4.11 are still $\mathcal{T}$-lemmas. Finally, since we are considering equality-interpolating theories, after step (4) of Algorithm 4.11 both C_i' and C_i'' are $\mathcal{T}$-lemmas.

(iii) Since the root of Π does not contain any interface equality (item (iii) of Definition 4.30), in step (5) $\neg\eta_i \equiv \neg\mu_i$ and $\neg\eta_k \equiv \neg\mu_k$, and therefore the root does not change.

Clearly, Algorithm 4.11 operates in linear time on the number of $\mathcal{T}$-lemmas, and thus of AB-mixed interface equalities. Moreover, every time an interface equality is split, only two new nodes are added to the proof (a right leaf and an inner node), and therefore the size of Π' is linear in that of Π.

The advantage of having ie-local proofs is that they ease significantly the process of eliminating AB-mixed interface equalities. First, since all the reasoning involving interface equalities is confined in Π^{ie} subproofs, only such subproofs – which typically constitute only a small fraction of the whole proof – need to be traversed and manipulated. Second, the simple structure of Π^{ie} subproofs allows for an efficient application of the rewriting process of steps (5) and (3), preventing any explosion in size of the proof. In fact, e.g., if in step (5) the right premise of the last step were instead the root of some subproof Π_i with C_i as a leaf, then two copies of Π'_i and Π''_i would be produced, in which each instance of $(a_i = b_i)$ must be replaced with $(a_i = t_i)$ and $(t_i = b_i)$ respectively.

Generating ie-local proofs in DTC

In this section we show how to implement a variant of DTC so that to generate ie-local proofs of unsatisfiability. For the sake of simplicity, we describe first a simplified algorithm which makes use of two distinct DPLL engines. We then describe how to avoid the need of a second DPLL engine with the use of a particular search strategy for DTC.

The simplified algorithm uses two distinct DPLL engines, a *main* one and an *auxiliary* one, which we shall call DPLL-1 and DPLL-2 respectively. Consider Figure 4.12, left. DPLL-1 receives in input the clauses of the

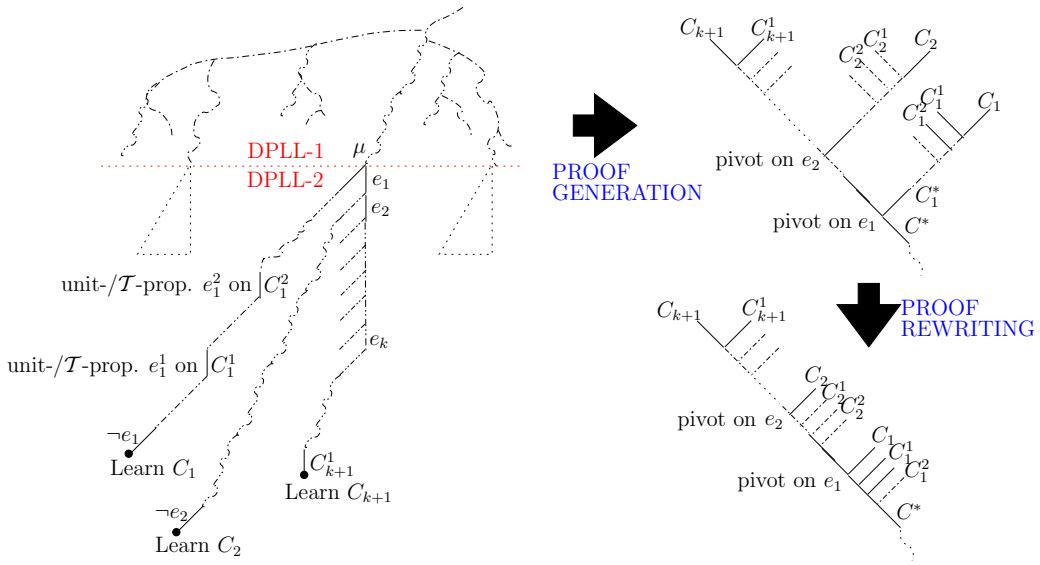


Figure 4.12: Simple strategy for generating ie-local proofs. Left: DTC search; top-right: corresponding (sub)proof; bottom-right: Π^{ie} (sub)proof after rewriting.

input problem ϕ (which we assume pure and $\mathcal{T}_1 \cup \mathcal{T}_2$-inconsistent), but no interface equality, which are instead given to DPLL-2. DPLL-1 enumerates total Boolean models μ of ϕ, and invokes the two $\mathcal{T}_i$-solvers separately on the subsets $\mu_{\mathcal{T}_1}$ and $\mu_{\mathcal{T}_2}$ of μ. If one $\mathcal{T}_i$-solver reports an inconsistency, then DPLL-1 backtracks. Otherwise, both $\mu_{\mathcal{T}_i}$ are $\mathcal{T}_i$-consistent, and DPLL-2 is invoked on the list of unit clauses composed of the $\mathcal{T}_1 \cup \mathcal{T}_2$-literals in μ, to check its $\mathcal{T}_1 \cup \mathcal{T}_2$-consistency.

DPLL-2 branches only on interface equalities, assigning them always to false first. Some interface equalities e_1^j, however, may be assigned to true by unit-propagation on previously-learned clauses in the form $C_1^j \stackrel{\text{def}}{=} \neg\mu_1^j \vee e_1^j$, or by $\mathcal{T}$-propagation on deduction clauses C_1^j in the same form; we call C_1^j the *antecedent clause* of e_1^j. [11] (As in [BCF+08], we assume that when a $\mathcal{T}$-propagation step $\mu_i^j \models_{\mathcal{T}} e_i^j$ occurs, μ_i^j being a subset of the current branch, the deduction clause $C_i^j \stackrel{\text{def}}{=} \neg\mu_i^j \vee e_i^j$ is learned, either temporarily or permanently; if so, we can see this step as a unit-propagation on C_i^j.) When all

[11] Notationally, e_i^j denotes the j-th most-recently unit-propagated interface equality in the branch in which C_i is learned, and $C_i^j \stackrel{\text{def}}{=} \neg\mu_i^j \vee e_i^j$ denotes the antecedent clause of e_i^j.

the interface equalities have been assigned a truth value, the propositional model $\mu' \equiv \mu_{\mathcal{T}_1} \cup \mu_{\mathcal{T}_2} \cup \mu_{\mathsf{ie}}$ is checked for $\mathcal{T}_1 \cup \mathcal{T}_2$-consistency by invoking each of the $\mathcal{T}_i$-solvers on $\mu_{\mathcal{T}_i} \cup \mu_{\mathsf{ie}}$. [12] Since ϕ is inconsistent, one of the two $\mathcal{T}_i$-solvers detects an inconsistency (if both do, we consider only the first). Therefore a $\mathcal{T}_i$-lemma C_1 is generated. As stated at the end of §4.5.1, we can assume without loss of generality that C_1 contains at most one positive interface equality e_1. (Notice also that all negative interface equalities $\neg e_1^j$ in C_1, if any, have been assigned by unit-propagation or $\mathcal{T}$-propagation on some antecedent clause C_1^j.) DPLL-2 then learns C_1 and uses it as conflicting clause to backjump: starting from C_1, it eliminates from the clause every $\neg e_1^j$ by resolving the current clause against its antecedent clause C_1^j, until no negated equality occurs in the final clause C_1^*. [13]

If C_1 includes one positive interface equality e_1, then also the final clause C_1^* includes it, so that DPLL-2 uses C_1^* as a conflict clause to jump up to μ and to unit-propagate e_1. Then DPLL-2 starts exploring a new branch. This process is repeated on several branches, learning a sequence of $\mathcal{T}$-lemmas $C_1, ..., C_k$ each C_i containing only one positive interface equality e_i, until a branch causes the generation of a $\mathcal{T}$-lemma C_{k+1} containing no positive interface equalities. Then C_{k+1} is resolved backward against the antecedent clauses of its negative interface equalities, generating a final conflict clause C^* which contains no interface equalities.

Overall, DPLL-2 has checked the $\mathcal{T}_1 \cup \mathcal{T}_2$-unsatisfiability of μ, building a resolution (sub)proof Π^* whose root is C^*. (Figure 4.12, top right.) Then the $\mathcal{T}_1 \cup \mathcal{T}_2$-lemma C^* is passed to DPLL-1, which uses it as a blocking clause for the assignment μ, it backtracks and continues the search. When

[12] In fact, it is not necessary to wait for all interface equalities to have a value before invoking the $\mathcal{T}_i$-solvers. Rather, the standard *early pruning* optimization (see §1.3.1 on page 18) can be applied.

[13] In order to determine the order in which to eliminate the interface equalities, the *implication graph* of the auxiliary DPLL engine can be used. This is a standard process in the conflict analysis in modern SAT and SMT solvers (see, e.g., [vG07, Seb07]).

the empty clause is obtained, it generates a proof of unsatisfiability in the usual way (see e.g. [vG07]).

Since the main solver knows nothing about interface equalities, they can only appear inside the proofs of the blocking clauses generated by the auxiliary solver (like Π^*). Each Π^* is not yet a Π^{ie} subproof, since it complies only with items (i), (ii) and (iii) of Definition 4.30 but not with item (iv). The reason for the latter fact is that Π^* contains a set of right branches Π_{C_i}, one of each $\mathcal{T}$-lemma C_i in $\{C_{k+1}, ..., C_1\}$, representing the resolution steps to resolve away the interface equalities introduced by unit-propagation/$\mathcal{T}$-propagation in each branch. Each such sub-branch Π_{C_i}, however, can be reduced to length one by moving downwards the resolution steps with the antecedent clauses $C_i^1, C_i^2, ...$ which C_i encounters in the branch. (Figure 4.12, bottom right.) This is done by recursively applying the following rewriting step to Π_{C_i}, until it reduces to the single clause C_i:

$$\dfrac{\dfrac{\vdots}{\neg e_i \vee \neg \mu_i'} \quad \boxed{\dfrac{\overbrace{\neg \mu_i^j \vee e_i^j}^{C_i^j} \quad \dfrac{C_i^{j-1} \qquad \dfrac{C_i^1 \quad C_i}{\vdots}}{\neg \mu_i'' \vee \neg e_i^j \vee e_i}}{\neg \mu_i^j \vee \neg \mu_i'' \vee e_i}}^{\Pi_{C_i}}}{\neg \mu_i' \vee \neg \mu_i^j \vee \neg \mu_i''} \quad \Longrightarrow$$

$$\dfrac{\dfrac{\dfrac{\vdots}{\neg e_i \vee \neg \mu_i'} \quad \boxed{\dfrac{C_i^{j-1} \qquad \dfrac{C_i^1 \quad C_i}{\vdots}}{\neg \mu_i'' \vee \neg e_i^j \vee e_i}}^{\Pi'_{C_i}}}{\neg \mu_i' \vee \neg \mu_i'' \vee \neg e_i^j} \qquad \overbrace{\neg \mu_i^j \vee e_i^j}^{C_i^j}}{\neg \mu_i' \vee \neg \mu_i^j \vee \neg \mu_i''} \tag{4.10}$$

As a result, each Π^* is transformed into a Π^{ie} subproof, so that the final proof is ie-local.

In an actual implementation, there is no need of having two distinct DPLL solvers for constructing ie-local proofs. In fact, we can obtain the same result by adopting a variant of the DTC Strategy 1 of [BCF+08]. We never select an interface equality for case splitting if there is some other unassigned atom, and we always assign false to interface equalities first. Moreover, we "delay" $\mathcal{T}$-propagation of interface equalities until all the original atoms have been assigned a truth value. Finally, when splitting on interface equalities, we restrict both the backjumping and the learning procedures of the DPLL engine as follows. Let d be the depth in the DPLL tree at which the first interface equality is selected for case splitting. If during the exploration of the current DPLL branch we have to backjump above d, then we generate by resolution a conflict clause that does not contain any interface equality, and "deactivate" all the $\mathcal{T}$-lemmas containing some interface equality — that is, we do not use such $\mathcal{T}$-lemmas for performing unit propagation — and we re-activate them only when we start splitting on interface equalities again. Using such strategy, we obtain the same effect as in the simple algorithm that uses two DPLL engines: the search space is partitioned in two distinct subspaces, the one of original atoms and the one of interface equalities, and the generated proof of unsatisfiability reflects such partition.

Finally, we remark that what described above is only *one* possible strategy for generating ie-local proofs, and not necessarily the most efficient one. Moreover, that of generating ie-local proofs is only a *sufficient* condition to obtain interpolants from DTC avoiding duplications of sub-proofs, and more general strategies may be conceived.

4.5.3 Discussion

Our new DTC-based combination method has several advantages over the traditional one of [YM05] based on N.O.:

1. It inherits all the advantages of DTC over the traditional N.O. in terms of versatility, efficiency and restrictions imposed to $\mathcal{T}$-solvers [BBC$^+$06b, BCF$^+$08]. Moreover, it allows for using a more modern SMT solver, since many state-of-the-art solvers adopt variants or extensions of DTC instead of N.O..

2. Instead of requiring an "ad-hoc" method for performing the combination, it exploits the Boolean interpolation algorithm. In fact, thanks to the fact that interface equalities occur in the proof of unsatisfiability Π, once the AB-mixed terms in Π are split there is no need of any interpolant-combination method at all. In contrast, with the N.O.-based method of [YM05] interpolants for $\mathcal{T}_1 \cup \mathcal{T}_2$-lemmas are generated by combining "theory-specific partial interpolants" for the two $\mathcal{T}_i$'s with an algorithm that essentially duplicates the work that in our case is performed by the Boolean algorithm. This allows also for potentially exploiting optimization techniques for Boolean interpolation which are or will be made available from the literature.

3. By splitting AB-mixed terms only *after* the construction of the proof Π, it allows for computing several interpolants for several different partitions of the input problem into (A, B) *from the same proof* Π . This is particularly important for applications in abstraction refinement [HJMM04]. (This feature is discussed in §4.5.4.)

The work of [YM05] can in principle deal with non-convex theories. Our approach is currently limited to the case of convex theories; however, we see no reason that would prevent it from being extensible at least theoretically to the case of nonconvex theories. We also remark that implementing the algorithm of [YM05] for non-convex theories is a non-trivial task, and in fact we are not aware of any such implementation.

Another algorithm for computing interpolants in combined theories is given in [SS08]. Rather than a combination of theories with disjoint signatures, that work considers the interpolation problem for extensions of a base (convex) theory with new function symbols, and it is therefore orthogonal to ours. The solution adopted is however similar to what we propose, in the sense that also the algorithm of [SS08] works by splitting AB-mixed terms. The difference is that our algorithm is tightly integrated in an SMT context, as it is guided by the resolution proof generated by the DPLL engine.

The recent work of [GKT09] proposes a generalization of our method that avoids the construction of ie-local proofs. Rather, a generalization of Algorithm 4.11 to arbitrary resolution proofs is described, by introducing additional proof-manipulation steps. The advantage of this method is that it does not impose any restriction – which could potentially have a negative impact on performance – to the search strategy of DPLL. Unfortunately however, this method might cause an exponential blowup in the size of the proof [GKT09], because it might require the (recursive) duplication of whole proof trees for eliminating AB-mixed interface equalities. In contrast, when using ie-local proofs, the exponential blowup is avoided (see also the discussion on page 161).

4.5.4 Generating multiple interpolants

In §4.1 we remarked that a sufficient condition for generating multiple interpolants is that all the interpolants I_i's are computed from the same proof of unsatisfiability.

When generating interpolants with our DTC-based algorithm, however, we generate a different proof of unsatisfiability Π_i for each partition of the input formula ϕ into A_i and B_i. In particular, every Π_i is obtained from the same "base" proof Π, by splitting all the A_iB_i-mixed interface equalities

with Algorithm 4.11. In this section, we show that (4.2) (at §4.1 on page 113) holds also when each Π_i is obtained from the same ie-local proof Π by the rewriting of Algorithm 4.11 of §4.5.2. In order to do so, we need the following lemma.

Lemma 4.34. *Let Θ be a $\mathcal{T}_1 \cup \mathcal{T}_2$-lemma, and let Π be a Π^{ie} proof for it which does not contain any AB-mixed term. Then the formula I_Θ associated to Θ in Algorithm 4.1 is an interpolant for $(\neg\Theta \setminus B, \neg\Theta \downarrow B)$.*

Proof. By induction on the structure of Π, we have to prove that:

1. $\neg\Theta \setminus B \models I_\Theta$;
2. $I_\Theta \wedge (\neg\Theta \downarrow B) \models \bot$;
3. I_Θ contains only common symbols.

The base case is when Π is just a single leaf. Then, the lemma trivially holds by definition of I_Θ in this case (see Algorithm 4.1).

For the inductive step, let $\Theta_1 \stackrel{\text{def}}{=} (x = y) \vee \phi_1$ and $\Theta_2 \stackrel{\text{def}}{=} \neg(x = y) \vee \phi_2$ be the antecedents of Θ in Π. (So $\Theta \stackrel{\text{def}}{=} \phi_1 \vee \phi_2$). Let I_{Θ_1} and I_{Θ_2} be the interpolants for Θ_1 and Θ_2 (by the inductive hypothesis).

- If $(x = y) \not\preceq B$, then $I_\Theta \stackrel{\text{def}}{=} I_{\Theta_1} \vee I_{\Theta_2}$.
  1. By the inductive hypothesis, $(\neg\phi_1 \wedge \neg(x = y)) \setminus B \equiv (\neg\phi_1 \setminus B) \wedge \neg(x = y) \models I_{\Theta_1}$, and $(\neg\phi_2 \setminus B) \wedge (x = y) \models I_{\Theta_2}$. Then by resolution $(\neg\phi_1 \wedge \neg\phi_2) \setminus B \equiv \neg\Theta \setminus B \models I_\Theta$.
  2. By the inductive hypothesis, $I_{\Theta_1} \models \phi_1 \downarrow B$ and $I_{\Theta_2} \models \phi_2 \downarrow B$, so $I_{\Theta_1} \vee I_{\Theta_2} \models (\phi_1 \vee \phi_2) \downarrow B$, that is $I_\Theta \wedge (\neg\Theta \downarrow B) \models \bot$.
  3. By the inductive hypothesis both I_{Θ_1} and I_{Θ_2} contain only common symbols, and so also I_Θ does.

- If $(x = y) \preceq B$, then $I_\Theta \stackrel{\text{def}}{=} I_{\Theta_1} \wedge I_{\Theta_2}$.

  1. By the inductive hypothesis, $\neg\phi_1 \setminus B \models I_{\Theta_1}$ and $\neg\phi_2 \setminus B \models I_{\Theta_2}$, so $(\neg\phi_1 \wedge \neg\phi_2) \setminus B \equiv \neg\Theta \setminus B \models I_\Theta$.
  2. By the inductive hypothesis, we also have that $I_{\Theta_1} \models \phi_1 \downarrow B \vee (x = y)$ and $I_{\Theta_2} \models \phi_2 \downarrow B \vee \neg(x = y)$. Therefore, $I_{\Theta_1} \wedge I_{\Theta_2} \models (\phi_1 \vee \phi_2) \downarrow B$, that is $I_\Theta \wedge (\neg\Theta \downarrow B) \models \bot$.
  3. Finally, also in this case both I_{Θ_1} and I_{Θ_2} contain only common symbols, and so also I_Θ does. □

We now formalize the sufficient condition of [HJMM04] that (4.2) holds if the I_i's are computed from the same Π. The proof of it will be useful for showing that (4.2) holds also if the I_i's are computed from Π_i's obtained from Π by splitting the A_iB_i-mixed interface equalities.

Theorem 4.35. *Let $\phi \stackrel{def}{=} \phi_1 \wedge \phi_2 \wedge \phi_3$, and let Π be a proof of unsatisfiability for it. Let $A' \stackrel{def}{=} \phi_1$, $B' \stackrel{def}{=} \phi_2 \wedge \phi_3$, $A'' \stackrel{def}{=} \phi_1 \wedge \phi_2$ and $B'' \stackrel{def}{=} \phi_3$, and let I' and I'' be two interpolants for (A', B') and (A'', B'') respectively, both computed from Π. Then*

$$I' \wedge \phi_2 \models I''.$$

Proof. Let Π_Θ be a proof whose root is the clause Θ. We will prove, by induction on the structure of Π_Θ, that

$$I'_\Theta \wedge \phi_2 \models I''_\Theta \vee (\Theta \setminus \phi_3),$$

where I_Θ is defined as in Algorithm 4.1. The validity of the theorem follows immediately, by observing that the root of Π is $\bot$.

We have to consider three cases:

1. The first is when Θ is an input clause. Then, we have three subcases:

(a) If $\Theta \in \phi_3$, then $I'_\Theta \stackrel{\text{def}}{=} \top$, $I''_\Theta \stackrel{\text{def}}{=} \top$ and $(\Theta \setminus \phi_3) \equiv \bot$, so the theorem holds.

(b) If $\Theta \in \phi_1$, then $I'_\Theta \stackrel{\text{def}}{=} (\Theta \downarrow (\phi_2 \cup \phi_3))$, $I''_\Theta \vee (\Theta \setminus \phi_3) \stackrel{\text{def}}{=} (\Theta \downarrow \phi_3) \vee (\Theta \setminus \phi_3) \equiv \Theta$, so the theorem holds also in this case.

(c) If $\Theta \in \phi_2$, then $I'_\Theta \wedge \phi_2 \equiv \phi_2$ and $I''_\Theta \vee (\Theta \setminus \phi_3) \equiv \Theta$, so again the implication holds.

2. The second is when Θ is a $\mathcal{T}$-lemma. In this case, we have that I'_Θ is an interpolant for $(\neg\Theta \setminus (\phi_2 \cup \phi_3), \neg\Theta \downarrow (\phi_2 \cup \phi_3))$ and I''_Θ is an interpolant for $(\neg\Theta \setminus \phi_3, \neg\Theta \downarrow \phi_3)$. Therefore, by the definition of interpolant, $(\neg\Theta \setminus (\phi_2 \cup \phi_3)) \models I'_\Theta$ and $(\neg\Theta \setminus \phi_3) \models I''_\Theta$. Therefore, $I'_\Theta \vee (\Theta \setminus (\phi_2 \cup \phi_3))$ and $I''_\Theta \vee (\Theta \setminus \phi_3)$ are valid clauses, and so the implication trivially holds.

3. In this case Θ is obtained by resolution from $\Theta_1 \stackrel{\text{def}}{=} \phi \vee p$ and $\Theta_2 \stackrel{\text{def}}{=} \psi \vee \neg p$. If $p \in \phi_1$ or $p \in \phi_3$, then by the inductive hypotheses that $I'_{\Theta_i} \wedge \phi_2 \models I''_{\Theta_i} \vee (\Theta_i \setminus \phi_3)$, we have that $I'_\Theta \wedge \phi_2 \models I''_\Theta \vee (\Theta \setminus \phi_3)$.

   If $p \in \phi_2$, then $I'_\Theta \stackrel{\text{def}}{=} I'_{\Theta_1} \wedge I'_{\Theta_2}$ and $I''_\Theta \stackrel{\text{def}}{=} I''_{\Theta_1} \vee I''_{\Theta_2}$. Again, by the inductive hypotheses $I'_\Theta \wedge \phi_2 \models I''_\Theta \vee (\Theta \setminus \phi_3)$ holds. □

Theorem 4.36. *Let $\phi \stackrel{def}{=} \phi_1 \wedge \phi_2 \wedge \phi_3$. Let $A' \stackrel{def}{=} \phi_1$, $A'' \stackrel{def}{=} \phi_1 \wedge \phi_2$, $B' \stackrel{def}{=} \phi_2 \wedge \phi_3$, and $B'' \stackrel{def}{=} \phi_3$. Let Π be a proof of unsatisfiability for ϕ, and let Π' and Π'' be obtained from Π by splitting all the $A'B'$-mixed and $A''B''$-mixed interface equalities respectively. Let I' be an interpolant for (A', B') computed from Π', and I'' be an interpolant for (A'', B'') computed from Π''. Then*

$$I' \wedge \phi_2 \models I''.$$

Proof. We observe that Π' and Π'' are identical except for some Π^{ie} subproofs that contained some mixed interface equalities. Then, we can

proceed as in Theorem 4.35, we just need to consider one more case, namely when Θ is a $\mathcal{T}_1 \cup \mathcal{T}_2$-lemma at the root of a Π^{ie} subproof. In this case, thanks to Lemma 4.34 we have the same situation as in the second case of the proof of Theorem 4.35, and so we can apply the same argument. □

Thus, due to Theorem 4.36, we can use our DTC-based interpolation method in the context of abstraction refinement without any modification: it is enough to remember the original proof Π, and compute the interpolant I_i from the proof Π_i obtained by splitting the A_iB_i-mixed terms in Π, for each partition of the input formula ϕ into A_i and B_i as in (4.1).

4.6 Experimental evaluation

All the techniques presented in the previous sections have been implemented within MathSAT. In this section, we experimentally evaluate our approach.

4.6.1 Description of the benchmark sets

We have performed our experiments on two different sets of benchmarks. The first is obtained by running the Blast software model checker [BHJM07] on some Windows device drivers; these are similar to those used in [RSS07]. This is one of the most important applications of interpolation in formal verification, namely abstraction refinement in the context of CEGAR. The problem represents an abstract counterexample trace, and consists of a conjunction of atoms. In this setting, the interpolant generator is called very frequently, each time with a relatively simple input problem.

The second set of benchmarks originates from the SMT-LIB [RT06], and is composed of a subset of the unsatisfiable problems used in recent SMT solvers competitions [SMT]. The instances have been converted to CNF and then split in two consistent parts of approximately the same size. The

Table 4.2: Comparison of execution times of MATHSAT, FOCI, CLP-PROVER and CSISAT on problems generated by BLAST.

Family	# of problems	MathSAT	Foci	CLP-prover	CSIsat
kbfiltr.i	64	0.16	0.36	1.47	0.17
diskperf.i	119	0.33	0.78	3.08	0.39
floppy.i	235	0.73	1.64	5.91	0.86
cdaudio.i	130	0.35	1.07	2.98	0.47

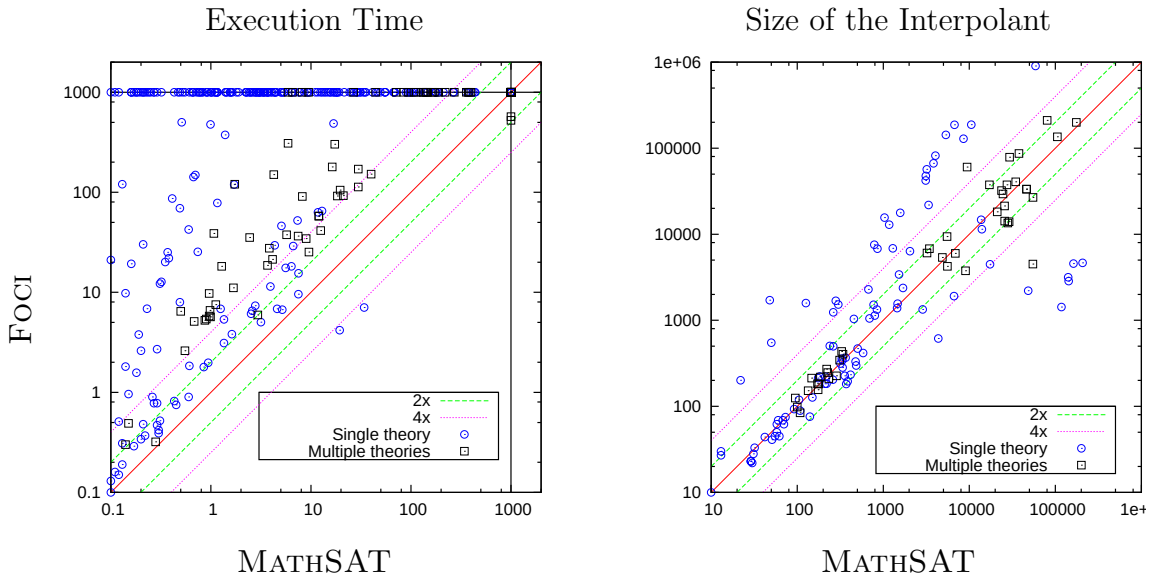


Figure 4.13: Comparison of MATHSAT and FOCI on SMT-LIB instances: execution time (left), and size of the interpolant (right). In the left plot, points on the horizontal and vertical lines are timeouts/failures.

set consists of problems of varying difficulty and with a nontrivial Boolean structure.

The experiments have been performed on a 3GHz Intel Xeon machine with 4GB of RAM running Linux. All the tools were run with a timeout of 600 seconds and a memory limit of 900 MB.

4.6.2 Comparison with the state-of-the-art tools available

In this section, we compare with the other interpolant generators which are available: FOCI [McM05, JM06], CLP-PROVER [RSS07] and CSISAT [BZM08].

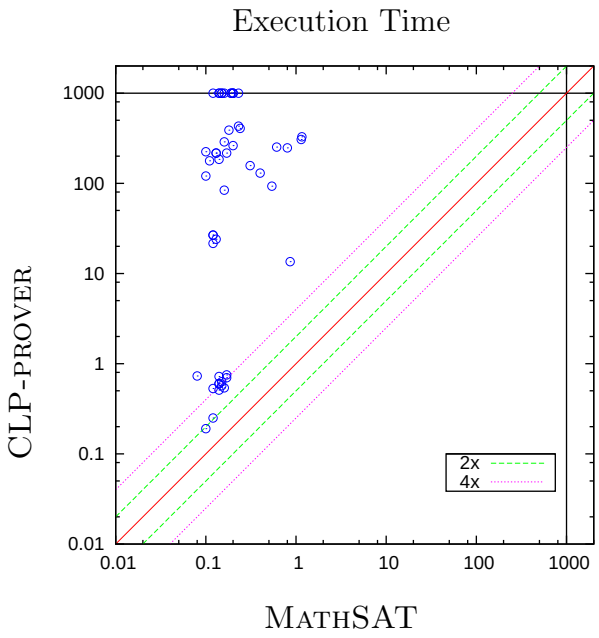


Figure 4.14: Comparison of MATHSAT and CLP-PROVER on conjunctions of $\mathcal{LA}(\mathbb{Q})$ atoms.

Other natural candidates for comparison would have been ZAP [BLM05] and LIFTER [KW07]; however, it was not possible to obtain them from the authors. We also remark that no comparison with INT2 [JCG08] is possible, since the domains of applications of MATHSAT and INT2 are disjoint: INT2 can handle $\mathcal{LA}(\mathbb{Z})$ equations/disequations and modular equations but only conjunctions of literals, whereas MATHSAT can handle formulae with arbitrary Boolean structure, but does not support $\mathcal{LA}(\mathbb{Z})$ except for its fragments $\mathcal{DL}(\mathbb{Z})$ and $\mathcal{UTVPI}(\mathbb{Z})$.

The comparison had to be adapted to the limitations of FOCI, CLP-PROVER and CSISAT. In fact, the current version of FOCI which is publicly available does not handle the full $\mathcal{LA}(\mathbb{Q})$, but only the $\mathcal{DL}(\mathbb{Q})$ fragment [14]. We also notice that the interpolants it generates are not always $\mathcal{DL}(\mathbb{Q})$ formulae. (See, e.g., Example 4.15 of §4.3.) CLP-PROVER does handle

[14] For example, it fails to detect the $\mathcal{LA}(\mathbb{Q})$-unsatisfiability of the following problem: $(0 \leq y - x + w) \wedge (0 \leq x - z - w) \wedge (0 \leq z - y - 1)$.

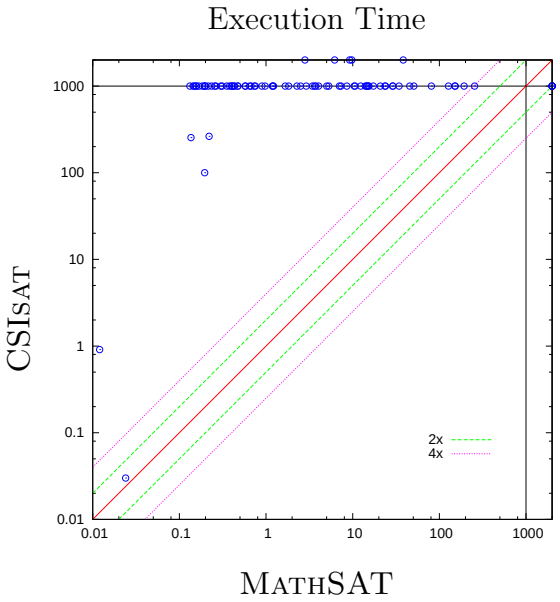


Figure 4.15: Comparison of MATHSAT and CSISAT on SMT-LIB instances.

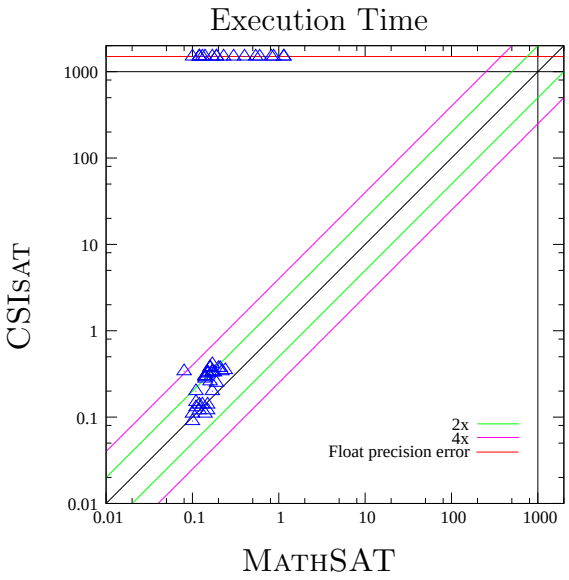


Figure 4.16: Comparison of MATHSAT and CSISAT on conjunctions of $\mathcal{LA}(\mathbb{Q})$ atoms.

the full $\mathcal{LA}(\mathbb{Q})$, but it accepts only conjunctions of atoms, rather than formulae with arbitrary Boolean structure. CSISAT, instead, can deal with $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q})$ formulae with arbitrary Boolean structure, but it does not support Boolean variables. These limitations made it impossible to compare all the four tools on all the instances of our benchmark sets. Therefore, we perform the following comparisons:

- We compare all the four solvers on the problems generated by BLAST;
- We compare MATHSAT with FOCI on SMT-LIB instances in the theories of $\mathcal{EUF}$, $\mathcal{DL}(\mathbb{Q})$ and their combination. In this case, we compare both the execution times and the sizes of the generated interpolants (in terms of number of nodes in the DAG representation of the formula). For computing interpolants in $\mathcal{EUF}$, we apply the algorithm of [McM05], using an extension of the algorithm of [NO07] to generate $\mathcal{EUF}$ proof trees (see §2.4). The combination $\mathcal{EUF} \cup \mathcal{DL}(\mathbb{Q})$ is

handled with the technique described in §4.5;

- We compare MathSAT, CLP-prover and CSIsat on $\mathcal{LA}(\mathbb{Q})$ problems consisting of conjunctions of atoms. These problems are single branches of the search trees explored by MathSAT for some $\mathcal{LA}(\mathbb{Q})$ instances in the SMT-LIB. We have collected several problems that took more than 0.1 seconds to MathSAT to solve, and then randomly picked 50 of them. In this case, we do not compare the sizes of the interpolants as they are always atomic formulae;
- We compare MathSAT and CSIsat on the subset (Consisting of 78 instances of the about 400 collected) of the SMT-LIB instances without Boolean variables.

The results are reported in Table 4.2 and in Figures 4.13, 4.14, 4.15 and 4.16. We can observe the following facts:

- Interpolation problems generated by Blast are trivial for all the tools. In fact, we even had some difficulties in measuring the execution times reliably. Despite this, MathSAT and CSIsat seem to be a little faster than the others.
- For problems with a nontrivial Boolean structure, MathSAT outperforms Foci in terms of execution time. This is true even for problems in the combined theory $\mathcal{EUF} \cup \mathcal{DL}(\mathbb{Q})$, despite the fact that the current implementation is still preliminary.

  As regards CSIsat, it could solve (within the time and memory limits) only 5 of the 78 instances it could potentially handle, and in all cases MathSAT outperforms it.
- In terms of size of the generated interpolants, the gap between MathSAT and Foci is smaller on average. However, the right plot of Figure 4.13 (which considers only instances for which both tools were

able to generate an interpolant) shows that there are more cases in which MATHSAT produces a smaller interpolant.

- On conjunctions of $\mathcal{LA}(\mathbb{Q})$ atoms, MATHSAT outperforms CLP-PROVER, sometimes by more than two orders of magnitude. The performance of MATHSAT and CSISAT is comparable on such instances, with MATHSAT being slightly faster. However, there are several cases in which CSISAT computes a wrong result, due to the use of floating-point arithmetic instead of infinite-precision arithmetic (which is used by MATHSAT – see §2.5.2).

4.6.3 Comparison between graph-based interpolation and interpolation in $\mathcal{LA}(\mathbb{Q})$

We conclude our experimental evaluation with a comparison between the general-purpose $\mathcal{LA}(\mathbb{Q})$ interpolation algorithm and the graph-based algorithm used for $\mathcal{DL}$ and $\mathcal{UTVPI}$, in order to demonstrate the usefulness of the specialized procedures of §4.3 and §4.4. For this comparison, we have randomly generated several $\mathcal{DL}(\mathbb{Q})$ and $\mathcal{UTVPI}(\mathbb{Q})$ interpolation problems of varying size and difficulty, and run both algorithms. The results are collected in Figure 4.17. The scatter plots show that the graph-based solver clearly outperforms the $\mathcal{LA}(\mathbb{Q})$ solver (sometimes by more than an order of magnitude), thus justifying the use of a specialized procedure. Furthermore, it can be seen that the computed interpolants, in addition to being within the theory of the input problem ($\mathcal{DL}(\mathbb{Q})$ or $\mathcal{UTVPI}(\mathbb{Q})$), are generally smaller, both in terms of nodes in the formula DAG and in number of atoms.

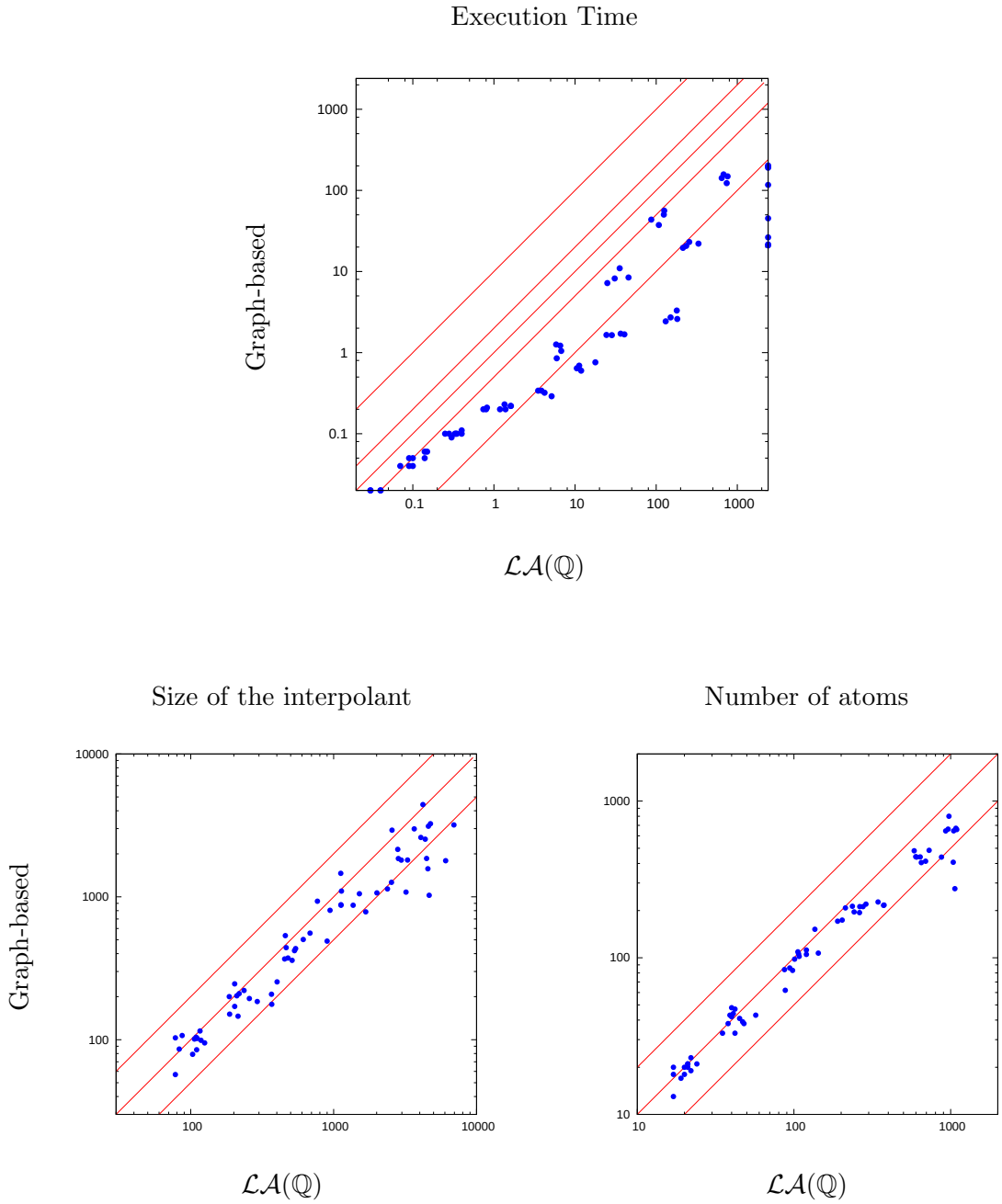


Figure 4.17: Comparison between graph-based and $\mathcal{LA}(\mathbb{Q})$ interpolation within MATHSAT.

Part III

Exploiting SMT for Software Verification

The availability of modern SMT solvers, capable of combining the high efficiency and scalability of propositional SAT solvers with the expressive power of first-order theories, is having a very big impact on formal verification. In the last few years, SMT solvers have been exploited in several novel verification techniques, among which e.g. [LNO06, CCF+07, ABM07, BH08, LQ08, AMP09, SDPK09, LQR09].

However, SMT is still a relatively new paradigm, and several popular verification techniques do not yet take full advantage of the power of modern SMT solvers. In the last part of our thesis, we concentrate on one such technique – ART-based software model checking [BHJM07] – and we show how to adapt it in order to better exploit the capability of a modern SMT solver like MathSAT of dealing efficiently with complex formulae. We empirically demonstrate that our technique leads to significant performance improvements on a standard set of benchmark C programs.

Chapter 5

Software Model Checking via Large-Block encoding

Note. *The work presented in this chapter was performed in collaboration with Dirk Beyer and Erkan Keremoglu of Simon Fraser University, Canada. Most of the material has already been presented in [BCG+09].*

Software model checking is an effective technique for software verification. Several advances in the field have lead to tools that are able to verify programs of considerable size, and show significant advantages over traditional techniques in terms of precision of the analysis. Prominent examples of such tools are the software model checkers SLAM [BR02] and BLAST [BHJM07]. However, efficiency and scalability remain major concerns in software model checking and hamper the adaptation of the techniques in industrial practice.

A successful approach to software model checking is based on the construction and analysis of an abstract reachability tree (ART), and predicate abstraction is one of the favorite abstract domains. The ART represents unwindings of the control-flow graph of the program. The search is usually guided by the control flow of the program. Nodes of the ART typically consist of the control-flow location, the call stack, and formulae that rep-

resent the data states. During the refinement process, the ART nodes are incrementally refined.

The construction and refinement of an ART require the use of decision procedures for several operations: for computing abstract successor states (by computing strongest postconditions of program operations), for checking entailment between formulae in the abstract space, for checking whether an abstract execution trace leading to an error is actually feasible in the concrete program, and for collecting information needed for abstraction refinement.

In the traditional ART approach, each program operation (assignment operation, assume operation, function call, function return) is represented by a single edge in the ART. Therefore, we call this approach *single-block encoding* (SBE). With SBE, each path in the ART represents a single (possibly infeasible) execution trace in the concrete program, and each node represents the result of following a single execution trace in the abstract space. Because of this, individual calls to decision procedures needed for performing the operations listed above are typically cheap, involving small formulae with a simple structure. However, a fundamental source of inefficiency of this approach is the fact that the control-flow of the program can induce a huge number of paths (and nodes) in the ART, which are explored independently of each other.

In order to overcome this problem, we propose a novel, broader view on ART-based software model checking, where a much more compact abstract space is used, resulting thus in a much smaller number of paths to be enumerated in the ART. Instead of using edges that represent single program operations, we encode entire parts of the program in one edge. In contrast to SBE, we call our new approach *large-block encoding* (LBE), which is enabled by and exploits the use of modern SMT techniques. In general, the new encoding may result in an exponential reduction of the

number of ART nodes.

The generalization from SBE to LBE has two main consequences. First, LBE requires a more general representation of abstract states than SBE. SBE is typically based on mere *conjunctions* of predicates. Because the LBE approach summarizes large portions of the control flow, conjunctions are not sufficient, and we need to use *arbitrary Boolean combinations* of predicates to represent the abstract states. Second, LBE requires a more accurate abstraction in the abstract-successor computations. Intuitively, an abstract edge represents many different paths of the program, and therefore it is necessary that the abstract-successor computations take the relationships between the predicates into account.

The practical effect of such two consequences of LBE is that the operations that decision procedures need to perform – for computing abstract successor states, for checking the feasibility of abstract traces in the concrete program, and for collecting information in order to refine the ART – become much more expensive, involving formulae that are significantly larger and more complex than with SBE. In other words, switching from SBE to LBE allows for moving the bottleneck of ART-based software model checking from the *number* of operations to be performed in the ART to their *cost*. This in turn allows to fully exploit the power and efficiency of modern SMT tools and techniques, which have shown to lead to significant scalability improvements over traditional approaches.

Contributions

We show that, by exploiting efficient SMT solvers, LBE leads to significant performance improvements to ART-based software model checking. We formally define LBE in terms of a summarization of the control-flow automaton for the program and we prove it correct. We analyze the differences between SBE and LBE in terms of the construction (computation of

abstract successors, refinement) and the exploration of the ART and of the interactions with the decision procedures used for implementing the basic steps of the verification algorithm. We implement the LBE approach, using MathSAT as a workhorse SMT engine, and we compare it both with our own implementation of SBE and with a state-of-the-art implementation of it represented by the software model checker Blast, on different benchmark programs commonly used in software model checking. The experiments show that our new approach outperforms the previous approach.

5.1 Background

5.1.1 Programs and Control-Flow Automata

We restrict the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all variables range over integers.[1] We represent a program by a *control-flow automaton* (CFA). A CFA $A \stackrel{\text{def}}{=} (L, G)$ consists of a set L of program locations, which model the program counter l, and a set $G \subseteq L \times Ops \times L$ of control-flow edges, which model the operations that are executed when control flows from one program location to another. The set of variables that occur in operations from Ops is denoted by X. A *program* $P \stackrel{\text{def}}{=} (A, l_0, l_E)$ consists of a CFA $A \stackrel{\text{def}}{=} (L, G)$ (which models the control flow of the program), an initial program location $l_0 \in L$ (which models the program entry) such that G does not contain any edge $(\cdot, \cdot, l_0)$, and a target program location $l_E \in L$ (which models the error location).

A *concrete data state* of a program is a variable assignment $c : X \to \mathbb{Z}$ that assigns to each variable an integer value. The set of all concrete data states of a program is denoted by $\mathcal{C}$. A set $r \subseteq \mathcal{C}$ of concrete data states

[1] Our implementation is based on CPAchecker [BK09], which operates on C programs that are given in the Cil intermediate language [NMRW02]; function calls are supported.

is called *region*. We represent regions using first-order formulae (with free variables from X): a formula ϕ represents the set S of all data states c that imply ϕ (i.e., $S = \{c \mid c \models \phi\}$). A *concrete state* of a program is a pair (l, c), where $l \in L$ is a program location and c is a concrete data state. A pair (l, ϕ) represents the following set of all concrete states: $\{(l, c) \mid c \models \phi\}$. The *concrete semantics* of an operation $op \in Ops$ is defined by the strongest postcondition operator SP_{op}: for a formula ϕ, $\mathsf{SP}_{op}(\phi)$ represents the set of data states that are reachable from any of the states in the region represented by ϕ after the execution of op. Given a formula ϕ that represents a set of concrete data states, for an assignment operation $s := e$, we have $\mathsf{SP}_{s:=e}(\phi) \stackrel{\text{def}}{=} \exists \widehat{s} : \phi_{[s \mapsto \widehat{s}]} \wedge (s = e_{[s \mapsto \widehat{s}]})$; and for an assume operation $assume(p)$, we have $\mathsf{SP}_{assume(p)}(\phi) \stackrel{\text{def}}{=} \phi \wedge p$.

A *path* σ is a sequence $\langle (op_1, l_1), ..., (op_n, l_n) \rangle$ of pairs of operations and locations. The path σ is called *program path* if for every i with $1 \leq i \leq n$ there exists a CFA edge $g \stackrel{\text{def}}{=} (l_{i-1}, op_i, l_i)$, i.e., σ represents a syntactical walk through the CFA. The *concrete semantics for a program path* $\sigma \stackrel{\text{def}}{=} \langle (op_1, l_1), ..., (op_n, l_n) \rangle$ is defined as the successive application of the strongest postoperator for each operation: $\mathsf{SP}_{\sigma}(\phi) \stackrel{\text{def}}{=} \mathsf{SP}_{op_n}(...\mathsf{SP}_{op_1}(\phi)...)$. The set of concrete states that result from running σ is represented by the pair $(l_n, \mathsf{SP}_{\sigma}(true))$. A program path σ is *feasible* if $\mathsf{SP}_{\sigma}(true)$ is satisfiable. A concrete state (l_n, c_n) is called *reachable* if there exists a feasible program path σ whose final location is l_n and such that $c_n \models \mathsf{SP}_{\sigma}(true)$. A location l is reachable if there exists a concrete state c such that (l, c) is reachable. A program is *safe* if l_E is not reachable.

5.1.2 Predicate Abstraction

Let $\mathcal{P}$ be a set of quantifier-free predicates over program variables in a theory $\mathcal{T}$. A *formula* ϕ is a Boolean combination of predicates from $\mathcal{P}$. A *precision for a formula* is a finite subset $\pi \subset \mathcal{P}$ of predicates. The

precision for a program is a function $\Pi : L \to 2^{\mathcal{P}}$, which assigns to each program location a precision for a formula.

Cartesian Predicate Abstraction

Let π be a precision. The *Cartesian predicate abstraction* $\phi^{\pi}_{\mathbb{C}}$ *of a formula* ϕ is the strongest conjunction of predicates from π that is entailed by ϕ: $\phi^{\pi}_{\mathbb{C}} \stackrel{\text{def}}{=} \bigwedge \{p \in \pi \mid \phi \implies p\}$. Such a predicate abstraction of a formula ϕ, which represents a region of concrete program states, is used as an *abstract state* (i.e., an abstract representation of the region) in program verification. For a formula ϕ and a precision π, the Cartesian predicate abstraction $\phi^{\pi}_{\mathbb{C}}$ of ϕ can be computed by $|\pi|$ SMT-solver queries. The abstract strongest postoperator SP^{π} for a predicate abstraction with precision π transforms the abstract state $\phi^{\pi}_{\mathbb{C}}$ into its successor $\phi'^{\pi}_{\mathbb{C}}$ for a program operation op, written as $\phi'^{\pi}_{\mathbb{C}} = \mathsf{SP}^{\pi}_{op}(\phi^{\pi}_{\mathbb{C}})$, if $\phi'^{\pi}_{\mathbb{C}}$ is the Cartesian predicate abstraction of $\mathsf{SP}_{op}(\phi^{\pi}_{\mathbb{C}})$, i.e., $\phi'^{\pi}_{\mathbb{C}} = (\mathsf{SP}_{op}(\phi^{\pi}_{\mathbb{C}}))^{\pi}_{\mathbb{C}}$. For more details, we refer the reader to the work of Ball et al. [BPR03].

Boolean Predicate Abstraction

Let π be a precision. The *Boolean predicate abstraction* $\phi^{\pi}_{\mathbb{B}}$ *of a formula* ϕ is the strongest Boolean combination of predicates from π that is entailed by ϕ. For a formula ϕ and a precision π, the Boolean predicate abstraction $\phi^{\pi}_{\mathbb{B}}$ of ϕ can be computed by querying an SMT solver in the following way: For each predicate $p_i \in \pi$, we introduce a propositional variable v_i. Now we ask an SMT solver to enumerate all satisfying assignments of $v_1, ..., v_{|\pi|}$ in the formula $\phi \wedge \bigwedge_{p_i \in \pi}(p_i \iff v_i)$. For each satisfying assignment, we construct a conjunction of all predicates from π whose corresponding propositional variable occurs positive in the assignment. The disjunction of all such conjunctions is the Boolean predicate abstraction for ϕ. The abstract strongest postoperator SP^{π} for a predicate abstraction with preci-

sion π transforms the abstract state $\phi^{\pi}_{\mathbb{B}}$ into its successor $\phi'^{\pi}_{\mathbb{B}}$ for a program operation *op*, written as $\phi'^{\pi}_{\mathbb{B}} = \mathsf{SP}^{\pi}_{op}(\phi^{\pi}_{\mathbb{B}})$, if $\phi'^{\pi}_{\mathbb{B}}$ is the Boolean predicate abstraction of $\mathsf{SP}_{op}(\phi^{\pi}_{\mathbb{B}})$, i.e., $\phi'^{\pi}_{\mathbb{B}} = (\mathsf{SP}_{op}(\phi^{\pi}_{\mathbb{B}}))^{\pi}_{\mathbb{B}}$. For more details, we refer the reader to the work of Lahiri et al. [LNO06].

5.1.3 ART-based Software Model Checking with SBE

An ART-based algorithm for software model checking takes an initial precision Π (which is typically very coarse) for the predicate abstraction, and constructs an ART for the input program and Π. An ART is a tree whose nodes are labeled with program locations and abstract states [BHJM07] (i.e., $n \stackrel{\text{def}}{=} (l, \phi)$). For a given ART node, all children nodes are labeled with successor locations and abstract successor states, according to the strongest postoperator and the predicate abstraction. A node $n \stackrel{\text{def}}{=} (l, \phi)$ is called *covered* if there exists another ART node $n' \stackrel{\text{def}}{=} (l, \phi')$ that entails n (that is, such that $\phi' \models \phi$). An ART is called *complete* if every node is either covered or all possible abstract successor states are present in the ART as children of the node. If a complete ART is constructed and the ART does not contain any error node, then the program is considered correct [BHJM07]. If the algorithm adds an error node to the ART, then the corresponding path σ is checked to determine if σ is feasible (that is, if the corresponding concrete program path is executable) or infeasible (that is, if there is no corresponding program execution). In the former case the path represents a witness for a program bug. In the latter case the path is analyzed, and a refinement Π' of Π is generated, such that the same path cannot occur again during the ART exploration. The concept of using an infeasible error path for abstraction refinement is called *counterexample-guided abstraction refinement (CEGAR)* [CGJ+03]. The concept of iteratively constructing an ART and refining only the precisions along the considered path is called *lazy abstraction* [HJMS02]. After re-

fining the precision, the algorithm continues with the next iteration, using Π' instead of Π to construct the ART, until either a complete error-free ART is obtained, or an error is found (note that the procedure might not terminate). For more details and a more in-depth illustration of the overall ART algorithm, we refer to the BLAST article [BHJM07].

A popular approach to extract predicates during refinement is to use interpolants [HJMM04]. Given a formula $\varphi \stackrel{\text{def}}{=} \phi_1 \wedge \ldots \wedge \phi_n$ corresponding to an infeasible path $\sigma \stackrel{\text{def}}{=} \langle (op_1, l_1), ..., (op_n, l_n) \rangle$, multiple interpolants $I_1, \ldots, I_{n-1}$ (see §4.1 on page 112) are computed for each partition of φ into $A_i \stackrel{\text{def}}{=} \phi_1 \wedge \ldots \wedge \phi_i$ and $B_i \stackrel{\text{def}}{=} \phi_{i+1} \wedge \ldots \wedge \phi_n$ for all i. Then, the refined precision Π' is built by adding, for each location l_i occurring in σ, all the atomic predicates occurring in the interpolant I_i to the current precision $\Pi(l_i)$. (For more details, see [HJMM04].)

In order to make the algorithm scale on practical examples, implementations such as BLAST or SLAM use the simple but coarse Cartesian abstraction, instead of the expensive but precise Boolean abstraction. Despite its potential imprecision, Cartesian abstraction has been proved successful for the verification of many real-world programs. In the SBE approach, given the large number of successor computations, the computation of the Boolean predicate abstraction is in fact too expensive, as it may require an SMT solver to enumerate an exponential number of assignments on the predicates in the precision, for each single successor computation. The reason for the success of Cartesian abstraction if used together with SBE, is that for a given program path, state overapproximations that are expressible as conjunctions of atomic predicates —for which Boolean and Cartesian abstractions are equivalent— are often good enough to prove that the error location is not reachable in the abstract space.

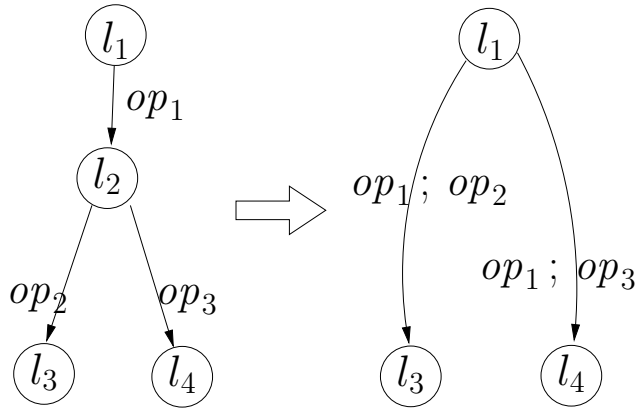
5.2 Large-Block Encoding

5.2.1 Summarization of Control-Flow Automata

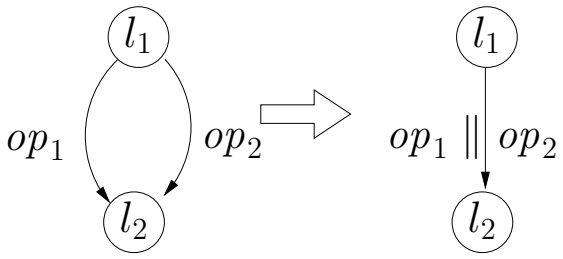
The large-block encoding is achieved by a summarization of the program CFA, in which each loop-free subgraph of the CFA is replaced by a single control-flow edge with a large formula that represents the removed subgraph. This process, which we call *CFA-summarization*, consists of the fixpoint application of the three rewriting rules that we describe below: first we apply Rule 0 once, and then we repeatedly apply Rules 1 and 2, until no rule is applicable anymore.
Let $P \stackrel{\text{def}}{=} (A, l_0, l_E)$ be a program with CFA $A \stackrel{\text{def}}{=} (L, G)$.

Rule 0 (Error Sink). We remove all edges $(l_E, \cdot, \cdot)$ from G, such that the target location l_E is a sink node with no outgoing edges.

Rule 1 (Sequence). If G contains an edge (l_1, op_1, l_2) with $l_1 \neq l_2$ and no other incoming edges for l_2 (i.e. edges $(\cdot, \cdot, l_2)$), and $G^{\rightarrow}_{l_2}$ is the subset of G of outgoing edges for l_2, then we change the CFA A in the following way: (1) we remove location l_2 from L, and (2) we remove the edge (l_1, op_1, l_2) and all edges in $G^{\rightarrow}_{l_2}$ from G, and for each edge $(l_2, op_i, l_i) \in G^{\rightarrow}_{l_2}$, we add the edge $(l_1, op_1\,;\, op_i, l_i)$ to G, where $\mathsf{SP}_{op_1\,;\,op_i}(\phi) \stackrel{\text{def}}{=} \mathsf{SP}_{op_i}(\mathsf{SP}_{op_1}(\phi))$. (Note that $G^{\rightarrow}_{l_2}$ might contain an edge $(l_2, \cdot, l_1)$.)



Rule 2 (Choice). If $L_2 \stackrel{\text{def}}{=} \{l_1, l_2\}$ and $A_{|L_2} \stackrel{\text{def}}{=} (L_2, G_2)$ is the subgraph



of A with nodes from L_2 and the set G_2 of edges contains the two edges (l_1, op_1, l_2) and (l_1, op_2, l_2), then we change the CFA A in the following way: (1) we remove the two edges (l_1, op_1, l_2) and (l_1, op_2, l_2) from G and add the edge $(l_1, op_1 \parallel op_2, l_2)$ to G, where $\mathsf{SP}_{op_1 \| op_2}(\phi) \stackrel{\text{def}}{=} \mathsf{SP}_{op_1}(\phi) \vee \mathsf{SP}_{op_2}(\phi)$. (Note that there might be a backwards edge $(l_2, \cdot, l_1)$.)

Let $P \stackrel{\text{def}}{=} (A, l_0, l_E)$ be a program and let A' be a CFA. The CFA A' is a *CFA-summary* of A if A' is obtained from A via an application of Rule 0 and then stepwise applications of Rules 1 and 2, and no rule can be further applied.

Example 5.1. Figure 5.1 shows a program (a) and its corresponding CFA (b). The control-flow automaton (CFA) is iteratively transformed to a CFA-summary (h) as follows: Rule 1 eliminates location 6 to (c), Rule 1 eliminates location 3 to (d), Rule 1 eliminates location 4 to (e), Rule 2 replaces the two edges 2–5 to (f), Rule 1 eliminates location 5 to (g), Rule 1 eliminates location 2 to (h). ◊

In the context of this thesis, we use the CFA-summary for program analysis, that is, we want to verify if the error location of the program is reachable. In order to prove that our summarization of a CFA is correct in this sense, we introduce some auxiliary lemmas.

Lemma 5.2. *Let (l, op, l') be a CFA edge, and $\{\varphi_i\}_i$ a collection of formulae. Then*

$$\mathsf{SP}_{op}(\bigvee_i \varphi_i) \equiv \bigvee_i \mathsf{SP}_{op}(\varphi_i).$$

```
L1: while (i>0) {
L2:   if (x==1) {
L3:     z = 0;
      } else {
L4:     z = 1;
      }
L5:   i = i-1;
L6: }
```

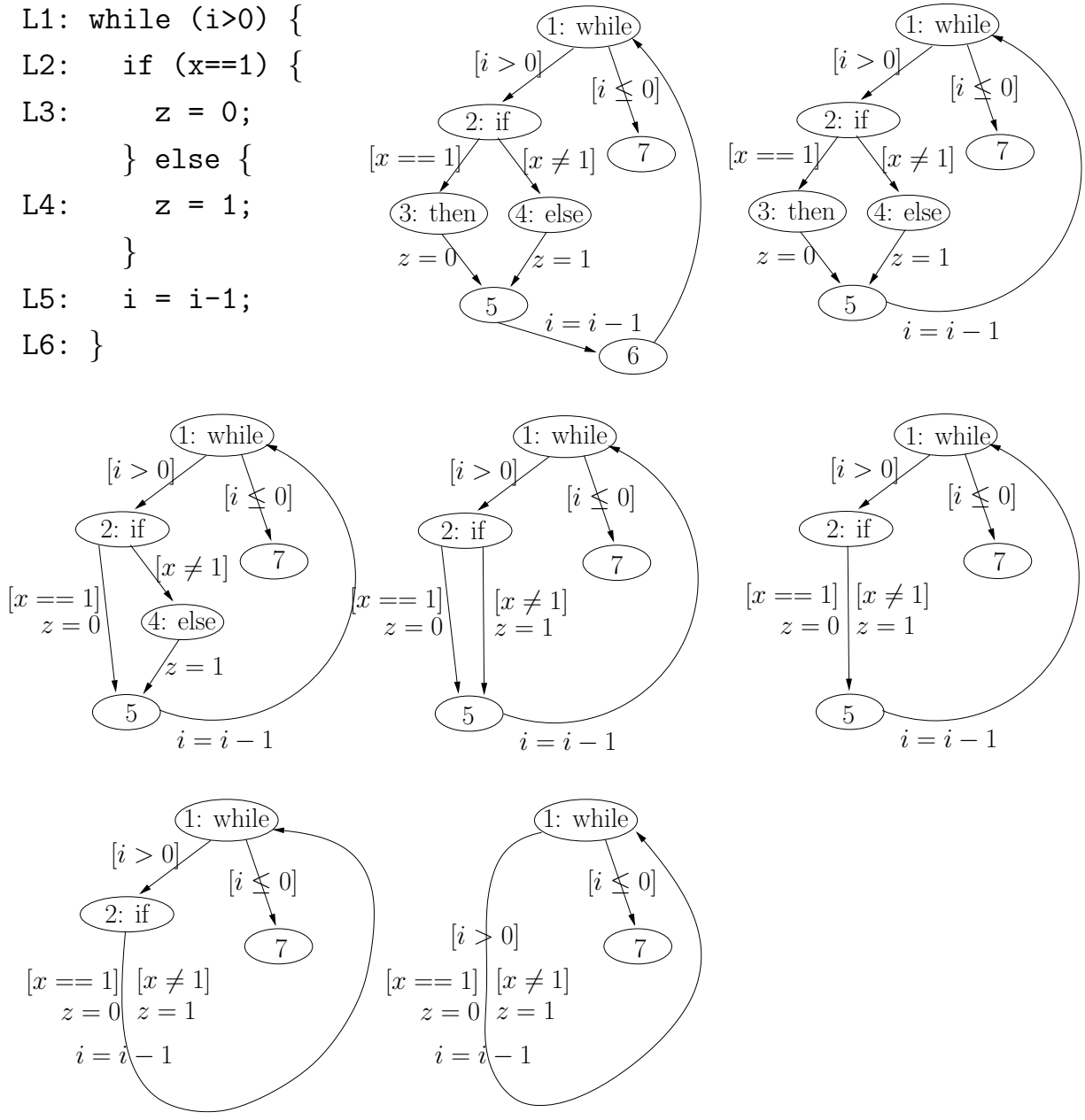


Figure 5.1: CFA Summarization: a) Program, b) CFA, c)–g) Intermediate CFAs, h) CFA-Summary. In the CFAs, $assume(p)$ is represented as $[p]$, $op_1 \,;\, op_2$ is represented by drawing op_2 below op_1, and $op_1 \parallel op_2$ by drawing op_2 beside op_1

Proof. If op is an assignment operation $s := e$, then

$$\begin{aligned}
\mathsf{SP}_{s:=e}(\bigvee_i \varphi_i) &= \exists \widehat{s}.((\bigvee_i \varphi_i)_{[s\mapsto\widehat{s}]} \wedge (s = e_{[s\mapsto\widehat{s}]})) \\
&\equiv \exists \widehat{s}.(\bigvee_i (\varphi_{i\,[s\mapsto\widehat{s}]} \wedge (s = e_{[s\mapsto\widehat{s}]}))) \\
&\equiv \bigvee_i (\exists \widehat{s}.(\varphi_{i\,[s\mapsto\widehat{s}]} \wedge (s = e_{[s\mapsto\widehat{s}]}))) \\
&\equiv \bigvee_i \mathsf{SP}_{s:=e}(\varphi_i)
\end{aligned}$$

If op is an assume operation $assume(p)$, then

$$\begin{aligned}
\mathsf{SP}_{assume(p)}(\bigvee_i \varphi_i) &= (\bigvee_i \varphi_i) \wedge p \\
&\equiv \bigvee_i (\varphi_i \wedge p) \\
&\equiv \bigvee_i \mathsf{SP}_{assume(p)}(\varphi_i)
\end{aligned}$$

The remaining two cases can be proved by induction.
If $op = op_1 \,;\, op_2$, then

$$\begin{aligned}
\mathsf{SP}_{op_1\,;\,op_2}(\bigvee_i \phi_i) &= \mathsf{SP}_{op_2}(\mathsf{SP}_{op_1}(\bigvee_i \phi_i)) \\
&\equiv \mathsf{SP}_{op_2}(\bigvee_i \mathsf{SP}_{op_1}(\phi_i)) \\
&\equiv \bigvee_i \mathsf{SP}_{op_2}(\mathsf{SP}_{op_1}(\phi_i)) \\
&\equiv \bigvee_i \mathsf{SP}_{op_1\,;\,op_2}(\phi_i)
\end{aligned}$$

If $op = op_1 \parallel op_2$, then

$$\begin{aligned}
\mathsf{SP}_{op_1\parallel op_2}(\bigvee_i \phi_i) &= \mathsf{SP}_{op_1}(\bigvee_i \phi_i) \vee \mathsf{SP}_{op_2}(\bigvee_i \phi_i) \\
&\equiv (\bigvee_i \mathsf{SP}_{op_1}(\phi_i)) \vee (\bigvee_i \mathsf{SP}_{op_2}(\phi_i)) \\
&\equiv \bigvee_i (\mathsf{SP}_{op_1}(\phi_i) \vee \mathsf{SP}_{op_2}(\phi_i)) \\
&\equiv \bigvee_i \mathsf{SP}_{op_1\parallel op_2}(\phi_i)
\end{aligned}$$

□

Lemma 5.3. *Let $A \stackrel{def}{=} (L, G)$ be a CFA, and let $A' \stackrel{def}{=} (L', G')$ be a summarization of A. Let σ be a path in A such that its initial and final locations occur also in L'. Then for all φ, there exists a path σ' in A', with the same initial and final locations as σ, such that $\mathsf{SP}_\sigma(\varphi) \models \mathsf{SP}_{\sigma'}(\varphi)$.*

Proof. CFA A' is obtained from A by a sequence of n rule applications. If $n = 0$ we have $A' = A$. If the lemma holds for one rule application, we can show by induction that the lemma holds for any finite sequence of rule applications.

We now show that the lemma holds for one rule application. Let $\sigma \stackrel{\text{def}}{=} \sigma_1, (l_i, op_i, l_j)$. The proof is by induction on the length of σ. (The base case is when σ_1 is empty.)

If $l_i \in L'$, by the inductive hypothesis there exists a path σ'_1 in A' such that $\mathsf{SP}_{\sigma_1}(\varphi) \models \mathsf{SP}_{\sigma'_1}(\varphi)$. If $(l_i, op_i, l_j) \in G'$, then we can take $\sigma' = \sigma'_1, (l_i, op_i, l_j)$. Otherwise, (l_i, op_i, l_j) must have been removed by an application of Rule 2, [2] and so G' contains an edge $(l_i, op_i \parallel \cdot, l_j)$. Therefore, we can take $\sigma' = \sigma'_1, (l_i, op_i \parallel \cdot, l_j)$.

If $l_i \notin L'$, then by hypothesis $\sigma \equiv \sigma_2, (l_k, op_k, l_i), (l_i, op_i, l_j)$. Moreover, l_i has been removed by an application of Rule 1. By the definition of Rule 1, (l_k, op_k, l_i) is the only incoming edge for l_i in G. Therefore, G' contains an edge $(l_k, op_k ; op_i, l_j)$ and clearly $l_k \in L'$. Thus, by the inductive hypothesis there exists a path σ'_2 in A' such that $\mathsf{SP}_{\sigma_2}(\varphi) \models \mathsf{SP}_{\sigma'_2}(\varphi)$, and so we can take $\sigma' = \sigma'_2, (l_k, op_k ; op_i, l_j)$. □

Lemma 5.4. *Let $A \stackrel{def}{=} (L, G)$ be a CFA, and let $A' \stackrel{def}{=} (L', G')$ be a summarization of A. Let σ' be a path in A'. Then for all φ, there exists a set Σ of paths in A, with the same initial and final locations as σ', such that $\mathsf{SP}_{\sigma'}(\varphi) \equiv \bigvee_{\sigma \in \Sigma} \mathsf{SP}_{\sigma}(\varphi)$.*

Proof. CFA A' is obtained from A by a sequence of n rule applications. If $n = 0$ we have $A' = A$. If the lemma holds for one rule application, we can show by induction that the lemma holds for any finite sequence of rule applications.

[2] It could not have been removed by Rule 1, because when Rule 1 removes the edges $(\cdot, \cdot, l)$ and $(l, \cdot, \cdot)$, it removes also the location l.

We now show that the lemma holds for one rule application. Let $\sigma' \stackrel{\text{def}}{=} \sigma'_p, (l_i, op_i, l_j)$ be a path in A'. The proof is by induction on the length of σ'. (The base case is when σ'_p is empty.)

First, we observe that all locations in σ' occur also in G.

By the inductive hypothesis, there exists a set Σ_p of paths in A, with the same initial and final locations as σ'_p, such that $\mathsf{SP}_{\sigma'_p}(\varphi) \equiv \bigvee_{\sigma_p \in \Sigma_p} \mathsf{SP}_{\sigma_p}(\varphi)$.

If $(l_i, op_i, l_j) \in G$, then we can take $\Sigma = \{\sigma_p, (l_i, op_i, l_j) \mid \sigma_p \in \Sigma_p\}$ (by Lemma 5.2).

Otherwise, (l_i, op_i, l_j) was generated by an application of one of the Rules. If it was generated by Rule 1, then G contains two edges (l_i, op'_i, l_k) and (l_k, op_k, l_j) such that $op_i = op'_i \,;\, op_k$. Then we can take $\Sigma = \{\sigma_p, (l_i, op'_i, l_k), (l_k, op_k, l_j) \mid \sigma_p \in \Sigma_p\}$ (by Lemma 5.2). If (l_i, op_i, l_j) was generated by Rule 2, then G contains two edges (l_i, op'_i, l_j) and (l_i, op''_i, l_j) such that $op_i = op'_i \parallel op''_i$. Let $\Sigma_1 = \{\sigma_p, (l_i, op'_i, l_j) \mid \sigma_p \in \Sigma_p\}$ and $\Sigma_2 = \{\sigma_p, (l_i, op''_i, l_j) \mid \sigma_p \in \Sigma_p\}$. Then we can take $\Sigma = \Sigma_1 \cup \Sigma_2$ (by Lemma 5.2). □

Now we can prove the correctness of our summarization.

Theorem 5.5 (Correctness of Summarization)**.** *Let $P \stackrel{def}{=} (A, l_0, l_E)$ be a program and let $A' \stackrel{def}{=} (L', G')$ be a CFA-summary of A. Then:*

(i) $\{l_0, l_E\} \subseteq L'$, and

(ii) l_E is reachable in (A', l_0, l_E) if and only if l_E is reachable in P.

Proof. Now we prove Theorem 5.5.

(i) The only Rule that removes locations is Rule 1. Since l_0 has no incoming edges (by definition) and l_E has no outgoing edges (because of Rule 0), they cannot be removed by Rule 1.

(ii) "$\rightarrow$" Follows from Lemma 5.3 and (i).
"$\leftarrow$" Follows from Lemma 5.4 and (i).

□

The summarization can be performed in polynomial time. The time taken by Rule 0 is proportional to the number of outgoing edges for l_E. Since each application of Rule 1 or Rule 2 removes at least one edge, there can be at most $|G|-1$ such applications. A naive way to determine the set of locations and edges to which to apply each rule requires $O(|V| \cdot k)$ time, where k is the maximum out-degree of locations. Finally, each application of Rule 2 requires $O(1)$ time, and each application of Rule 1 $O(k)$ time. Therefore, a naive summarization algorithm requires $O(|G| \cdot |V| \cdot k)$ time, which reduces to $O(|G| \cdot |V|)$ if k is bounded (that is, if we rewrite a priori all `switch`es into nested `if`s).

5.2.2 LBE versus SBE for Software Model Checking

The use of LBE instead of the standard SBE requires no modification to the general model-checking algorithm, which is still based on ART construction with CEGAR-based refinement. The main difference is that in LBE there is no one-to-one correspondence between ART paths and syntactical program paths. A single CFA edge corresponds to a *set of paths* between its source and target location, and a single ART path corresponds to a *set of program paths*. An ART node represents an overapproximation of the data region that is reachable by following *any* of the program paths represented by the ART path that leads to it. This difference leads to two observations.

First, LBE can lead to exponentially-smaller ARTs than SBE, and thus it can drastically reduce the number of successor computations (see Example 5.6) and the number of abstraction-refinement steps for infeasible error paths. Each of these operations, however, is typically more expensive than with SBE, because more complex formulae are involved.

Second, LBE requires a more general representation of abstract states.

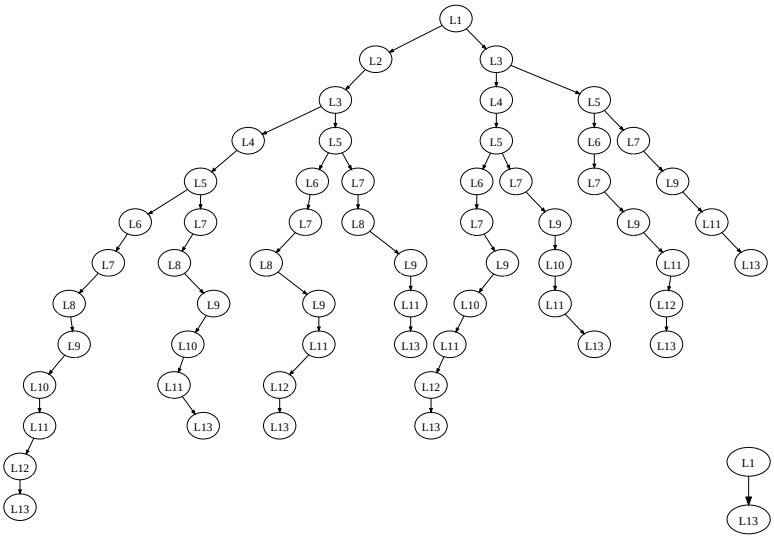
When using SBE, abstract states are typically represented as sets/conjunctions of predicates. This is sufficient for practical examples because each abstract state represents a data region reachable by a single program path, which can be encoded essentially as a conjunction of atomic formulae. With LBE, such representation would be too coarse, since each abstract state represents a data region that is reachable on several different program paths. Therefore, we need to use a representation for arbitrary (and larger) Boolean combinations of predicates. This generalization of the representation of abstract states requires a generalization of the representation of the transfers, that is, replacing the Cartesian abstraction with a more precise form of abstraction. In this thesis, we evaluate the use of the Boolean abstraction, which allows for a precise representation of arbitrary Boolean combinations of predicates.

With respect to the traditional SBE approach, LBE allows us to trade part of the cost of the *explicit* enumeration of program paths with that of the *symbolic* computation of abstract successor states: rather than having to build large ARTs via SBE by performing a substantial amount of relatively cheap operations (Cartesian abstract postoperator applications along single-block edges and counterexample analysis of individual program paths), we build smaller ARTs via LBE by performing more expensive symbolic operations (Boolean abstract postoperator applications along large portions of the control flow and counterexample analysis of multiple program paths), involving formulae with a complex Boolean structure. With LBE, the *cost* of each symbolic operation, rather than their *number*, becomes a critical performance factor.

To this extent, LBE makes it possible to fully exploit the power and functionality of modern SMT solvers: First, the capability of modern SMT solvers to perform large amounts of Boolean reasoning allows for handling large Boolean combinations of atomic expressions, instead of simple con-

```
L1:  if(p1) {
L2:    x1 = 1;
     }
L3:  if(p2) {
L4:    x2 = 2;
     }
L5:  if(p3) {
L6:    x3 = 3;
     }
L7:  if(p1) {
L8:    if (x1 != 1) goto ERR;
     }
L9:  if (p2) {
L10:   if (x2 != 2) goto ERR;
     }
L11: if (p3) {
L12:   if (x3 != 3) goto ERR;
     }
L13: return EXIT_SUCCESS;
ERR: return EXIT_FAILURE;
```

(a) Example C program



(b) ART for SBE

(c) ART for LBE

Figure 5.2: Example program and corresponding ARTs for SBE and LBE.

junctions. Second, the capability of some SMT solvers to perform All-SMT and interpolation allows for efficient computation of Boolean abstractions and interpolants, respectively. SMT-based Boolean abstraction and interpolation were shown to outperform previous approaches (see §4.6 and [LNO06, CCF+07]), especially when dealing with complex formulae. With SBE, instead, the use of modern SMT technology does not lead to significant improvements of the overall ART-based algorithm, because each SMT query involves only simple conjunctions. [3]

Example 5.6. We illustrate the advantage of LBE over SBE on the example program in Fig. 5.2 (a). In SBE, each program location is modeled explic-

[3]For example, BLAST uses SIMPLIFY, version 1.5.4, as of October 2001, for computing abstract successor states. Experiments have shown that replacing this old SIMPLIFY version by a highly-tuned modern SMT solver does not significantly improve the performance, because BLAST does not use much power of the SMT solver. Moreover, although MATHSAT outperforms other tools in the computation of Craig interpolants for general formulae, the difference in performance is negligible on formulae generated by a standard SBE ART-based algorithm (see Table 4.2 on page 172).

itly, and an abstract-successor computation is performed for each program operation. Figure 5.2 (b) shows the structure of the resulting ART. In the figure, abstract states are drawn as ellipses, and labeled with the location of the abstract state; the arrows indicate that there exists an edge from the source location to the target location in the control-flow. The ART represents all feasible program paths. For example, the leftmost program path is taking the 'then' branch of every 'if' statement. For every edge in the ART, an abstract-successor computation is performed, which potentially includes several SMT solver queries. The problems given to the SMT solver are usually very small, and the runtime sums up over a large amount of simple queries. Therefore, model checkers that are based on SBE (like Blast) experience serious performance problems on programs with such an exploding structure (see also the `test_locks` examples in Table 5.1). In LBE, the control-flow graph is summarized, such that control-flow edges represent entire subgraphs of the original control-flow. In our example, most of the program is summarized into one control-flow edge. Figure 5.2 (c) shows the structure of the resulting ART, in which all the feasible paths of the program are represented by a single edge. The exponential growth of the ART does not occur. ◊

5.3 Related Work

The model checkers Slam and Blast are typical examples for the SBE approach [BR02, BHJM07], both based on counterexample-guided abstraction refinement (CEGAR) [CGJ+03]. The CEGAR approach is followed also by the SatAbs tool [CKSY04]. Unlike Blast, SatAbs does not work by constructing an ART using lazy abstraction. Rather, it abstracts the input program into a Boolean program – a finite-state system whose variables represent truth-values of some predicates [BPR03] – and then checks

it using a standard symbolic model checker (e.g. [McM92, CCG+02]). The Boolean program is built by computing the Boolean abstraction of each basic block of the input program with respect to the current set of predicates. In this sense, therefore, this approach can also be seen as a form of SBE. However, differently from ART-based approaches, SATABS performs a fully symbolic search in the abstract space. Explicit search on an ART using lazy abstraction is instead adopted by McMillan in [McM06]. However, rather than using predicate abstraction for the abstract domain, Craig interpolants from infeasible error paths are directly used, thus avoiding abstract-successor computations.

A fundamentally different approach to software model checking is bounded model checking (BMC), with the most prominent example CBMC [CKL04]. Programs are unrolled up to a given depth, and a formula is constructed which is satisfiable if and only if one of the considered program executions reaches a certain error location. The BMC approaches are targeted towards discovering bugs, and can not be used to prove program safety.

Finally, the summarizations performed in our large-block encoding bear some similarities with the generation of verification conditions as performed by static program verifiers like SPEC# [BL05] or CALYSTO [BH08].

5.4 Experimental evaluation

In order to evaluate the proposed verification method, we integrate our algorithm as a new component into the configurable software verification toolkit CPACHECKER [BK09], using MATHSAT as the workhorse SMT engine. This implementation is written in JAVA. All example programs are preprocessed and transformed into the simple intermediate language CIL [NMRW02]. For parsing C programs, CPACHECKER uses a library from the Eclipse C/C++ Development Kit. We use binary decision dia-

grams (BDDs) for the representation of abstract-state formulae.

We run all experiments on a 2.66 GHz Intel Xeon machine with 16 GB of RAM and 6 MB of cache, running Linux. We used a timeout of 1 800 s and a memory limit of 2 GB.

5.4.1 Description of the benchmark programs

We use three categories of benchmark programs. First, we experiment with programs that are specifically designed to cause an exponential blowup of the ART when using SBE (`test_locks*`, in the style of Example 5.6). Second, we use the device-driver programs that were previously used as benchmarks in the Blast project. [4] Third, we solve various verification problems for the SSH client and server software (`s3_clnt*` and `s3_srvr*`), which share the same program logic, but check different safety properties. The safety property is encoded as conditional calls of a failure location and therefore reduces to the reachability of a certain error location. All benchmarks programs from the Blast web page are preprocessed with Cil. For the second and third groups of programs, we also performed experiments with artificial defects introduced.

5.4.2 Comparison with Blast

For a careful and fair performance comparison, we run experiments on three different configurations. First, we use Blast [BHJM07], version 2.5, which is a highly optimized state-of-the-art software model checker. Blast is implemented in the programming language OCaml. We run Blast using all four combinations of breadth-first search (`-bfs`) versus depth-first search (`-dfs`), both with and without heuristics for improving the predicate

[4] The Blast distribution contains 8 windows driver benchmarks. However, we could not run three of them (`parclass.i`, `mouclass.i` and `serial.i`), as Cil fails to parse them, making both CPAchecker and Blast fail.

discovery. BLAST provides five different levels of heuristics for predicate discovery, and we use only the lowest (`-predH 0`) and the highest option (`-predH 7`). Interestingly, every combination is best for some particular example programs, with considerable differences in runtime and memory consumption. The configuration using `-dfs -predH 7` is the winner (in terms of solved problems and total runtime) for the programs without defects, but is not able to verify three example programs (timeout). In the performance table, we provide results obtained using this configuration (column `-dfs -predH 7`), and also the best result among the four configurations for every single instance (column `best result`). For the unsafe programs, `-bfs -predH 7` performs best. All four configurations use the command-line options `-craig 2 -nosimplemem -alias ""`, which specify that BLAST runs with lazy, Craig-interpolation-based refinement, no CIL preprocessing for memory access, and without pointer analysis.

Second, in order to separate the optimization efforts in BLAST from the conceptual essence of the traditional lazy abstraction algorithm, we developed a re-implementation of the traditional algorithms as described in the BLAST tool article [BHJM07]. This re-implementation is integrated as component into CPACHECKER, so that the difference between SBE and LBE is only in the algorithms, not in the environment (same parser, same BDD package, same query optimization, etc.). Our SBE implementation uses a DFS algorithm. This column is labeled as SBE.

Third, we run the experiments using our new LBE algorithm, which is also implemented within CPACHECKER. Our LBE implementation uses a DFS algorithm. This column is labeled as LBE. Note that the purpose of our experiments is to give evidence of the performance difference between SBE and LBE, because these two settings are closest to each other, since SBE and LBE differ only in the CFA summarization and Boolean abstraction. The other two columns are provided to give evidence that the new

approach beats the highly optimized traditional implementation BLAST.

We actually configured and ran experiments with all four combinations: SBE versus LBE, and Cartesian versus Boolean abstraction. The experimentation clearly showed that SBE does not benefit from Boolean abstraction in terms of precision, with substantial degrade in performance: the only programs for which it terminated successfully were the first five instances of the `test_locks` group. Similarly, the combination of LBE with Cartesian abstraction fails to solve any of the experiments, due to loss of precision. Thus, we report only on the two successful configurations, that is, SBE in combination with Cartesian abstraction, and LBE in combination with Boolean abstraction.

5.4.3 Discussion of results

Tables 5.1 and 5.2 present performance results of our experiments, for the safe and unsafe programs respectively. All runtimes are given in seconds of processor time, '>1800.00' indicates a timeout, 'MO' indicates an out-of-memory error and 'NP' an error due to the impossibility of discovering new predicates during refinement in order to rule-out the spurious counterexample found. Table 5.3 shows statistics about the algorithms for SBE and LBE only.

The first group of experiments in Table 5.1 shows that the time complexity of SBE (and BLAST) can grow exponentially in the number of nested conditional statements, as expected. Table 5.3 explains why the SBE approach does not scale: the number of abstract nodes in the reachability tree grows exponentially in the number of predicates. The LBE approach reduces the loop-free part of the branching control-flow structure to a few edges (see Example 5.6), and the size of the ART is constant, because only the structure inside the body of the loop changes. There are no refinement steps necessary in the LBE approach, because the edges to the error

Table 5.1: Comparison between BLAST and CPACHECKER (SBE and LBE) on safe programs. 'MO' indicates an "Out of Memory" error, and 'NP' a "No new predicates found during refinement" error.

	Blast		**CPAchecker**	
Program	`(best result)`	`(-dfs -predH 7)`	**SBE**	**LBE**
test_locks_5.c	1.39	1.41	1.43	**0.10**
test_locks_6.c	2.48	2.58	2.51	**0.11**
test_locks_7.c	4.41	4.58	4.55	**0.12**
test_locks_8.c	8.24	8.48	8.92	**0.13**
test_locks_9.c	16.07	16.51	19.08	**0.14**
test_locks_10.c	33.68	34.29	53.88	**0.14**
test_locks_11.c	75.93	76.66	201.23	**0.15**
test_locks_12.c	186.36	187.86	MO	**0.15**
test_locks_13.c	523.81	523.81	MO	**0.16**
test_locks_14.c	1738.60	1738.60	MO	**0.17**
test_locks_15.c	>1800.00	>1800.00	MO	**0.18**
cdaudio.i.cil.c	69.87	101.60	NP	**16.84**
diskperf.i.cil.c	NP	>1800.00	NP	**37.38**
floppy.i.cil.c	87.17	>1800.00	NP	**18.03**
kbfiltr.i.cil.c	8.68	11.85	14.25	**2.66**
parport.i.cil.c	305.55	346.52	NP	**134.21**
s3_clnt.blast.01.i.cil.c	13.27	516.54	207.57	**3.09**
s3_clnt.blast.02.i.cil.c	25.56	131.66	428.42	**8.51**
s3_clnt.blast.03.i.cil.c	24.74	131.63	293.16	**3.07**
s3_clnt.blast.04.i.cil.c	26.54	81.33	285.59	**3.98**
s3_srvr.blast.01.i.cil.c	388.27	490.73	1676.10	**52.21**
s3_srvr.blast.02.i.cil.c	158.00	158.00	>1800.00	**17.94**
s3_srvr.blast.03.i.cil.c	115.96	115.96	>1800.00	**28.62**
s3_srvr.blast.04.i.cil.c	72.45	128.55	MO	**23.45**
s3_srvr.blast.06.i.cil.c	131.19	131.19	>1800.00	**10.92**
s3_srvr.blast.07.i.cil.c	214.89	324.22	>1800.00	**50.21**
s3_srvr.blast.08.i.cil.c	**46.08**	**46.08**	>1800.00	255.27
s3_srvr.blast.09.i.cil.c	196.07	549.36	>1800.00	**135.58**
s3_srvr.blast.10.i.cil.c	**46.24**	**46.24**	>1800.00	77.01
s3_srvr.blast.11.i.cil.c	160.81	440.06	>1800.00	**15.54**
s3_srvr.blast.12.i.cil.c	130.81	130.81	>1800.00	**7.15**
s3_srvr.blast.13.i.cil.c	256.79	428.06	>1800.00	**80.20**
s3_srvr.blast.14.i.cil.c	131.26	131.26	>1800.00	**31.61**
s3_srvr.blast.15.i.cil.c	46.34	46.34	>1800.00	**4.81**
s3_srvr.blast.16.i.cil.c	130.88	130.88	>1800.00	**34.40**
TOTAL (solved/time)	**33 / 5378.39**	**32 / 7213.65**	**13 / 3196.69**	**35 / 1054.24**
TOTAL w/o `test_locks*`	**23 / 2787.42**	**22 / 4618.87**	**6 / 2905.09**	**24 / 1052.69**

Table 5.2: Comparison between Blast and CPAchecker (SBE and LBE) on programs with artificial bugs. 'MO' indicates an "Out of Memory" error, and 'NP' a "No new predicates found during refinement" error.

	Blast		**CPAchecker**	
Program	`(best result)`	`(-bfs -predH 7)`	**SBE**	**LBE**
cdaudio.BUG.i.cil.c	6.85	36.74	24.51	**2.67**
diskperf.BUG.i.cil.c	333.72	>1800.00	7.38	**2.26**
floppy.BUG.i.cil.c	44.45	1035.71	12.37	**1.46**
kbfiltr.BUG.i.cil.c	17.10	53.28	26.73	**3.87**
parport.BUG.i.cil.c	**0.61**	4.01	4.65	0.78
s3_clnt.blast.01.BUG.i.cil.c	3.18	11.06	466.06	**1.84**
s3_clnt.blast.02.BUG.i.cil.c	3.63	3.63	326.14	**0.90**
s3_clnt.blast.03.BUG.i.cil.c	2.64	2.64	300.59	**1.32**
s3_clnt.blast.04.BUG.i.cil.c	4.01	4.01	284.19	**1.31**
s3_srvr.blast.01.BUG.i.cil.c	3.79	3.79	>1800.00	**0.78**
s3_srvr.blast.02.BUG.i.cil.c	2.94	2.94	>1800.00	**0.70**
s3_srvr.blast.03.BUG.i.cil.c	3.04	3.04	>1800.00	**0.74**
s3_srvr.blast.04.BUG.i.cil.c	2.99	2.99	>1800.00	**0.72**
s3_srvr.blast.06.BUG.i.cil.c	15.45	22.24	719.64	**1.22**
s3_srvr.blast.07.BUG.i.cil.c	133.34	133.34	>1800.00	**1.73**
s3_srvr.blast.08.BUG.i.cil.c	15.37	29.56	749.61	**3.24**
s3_srvr.blast.09.BUG.i.cil.c	111.60	111.60	MO	**3.64**
s3_srvr.blast.10.BUG.i.cil.c	15.29	26.57	736.58	**3.29**
s3_srvr.blast.11.BUG.i.cil.c	19.21	19.21	MO	**0.96**
s3_srvr.blast.12.BUG.i.cil.c	15.13	15.13	1420.56	**1.57**
s3_srvr.blast.13.BUG.i.cil.c	105.12	105.12	>1800.00	**1.09**
s3_srvr.blast.14.BUG.i.cil.c	15.33	21.25	632.27	**7.18**
s3_srvr.blast.15.BUG.i.cil.c	15.39	31.26	999.84	**3.47**
s3_srvr.blast.16.BUG.i.cil.c	15.47	21.95	760.48	**1.66**
TOTAL (solved/time)	**24 / 905.65**	**23 / 1701.07**	**16 / 7471.59**	**24 / 48.40**

location are infeasible. Therefore, no predicates are used. The runtime of the LBE approach slightly increases with the size of the program, because the formulae that are sent to the SMT solver are slightly increasing. Although in principle the complexity of the SMT problem grows exponentially in the size of the formulae, the heuristics used by SMT solvers avoid the exponential enumeration that we observe in the case of SBE.

For the two other classes of experiments, we see that LBE is able to

Table 5.3: Detailed comparison between LBE and SBE encodings; entries marked with (*) denote partial statistics for analyses that terminated unsuccessfully (if available).

	LBE					SBE				
	ART	# ref	# predicates			ART	# ref	# predicates		
Program	size	steps	Tot	Avg	Max	size	steps	Tot	Avg	Max
test_locks_5.c	4	0	0	0	0	1344	50	10	3	10
test_locks_6.c	4	0	0	0	0	2301	72	12	4	12
test_locks_7.c	4	0	0	0	0	3845	98	14	5	14
test_locks_8.c	4	0	0	0	0	6426	128	16	6	16
test_locks_9.c	4	0	0	0	0	10926	162	18	7	18
test_locks_10.c	4	0	0	0	0	19091	200	20	8	20
test_locks_11.c	4	0	0	0	0	34395	242	22	9	22
test_locks_12.c	4	0	0	0	0	45596(*)	288(*)	24(*)	10(*)	24(*)
test_locks_13.c	4	0	0	0	0	–	–	–	–	–
test_locks_14.c	4	0	0	0	0	–	–	–	–	–
test_locks_15.c	4	0	0	0	0	–	–	–	–	–
cdaudio.i.cil.c	7138	124	79	5	16	17769(*)	200(*)	83(*)	11(*)	62(*)
diskperf.i.cil.c	4184	120	64	6	23	18336(*)	180(*)	83(*)	12(*)	53(*)
floppy.i.cil.c	9471	167	55	4	13	52169(*)	531(*)	158(*)	9(*)	53(*)
kbfiltr.i.cil.c	1576	47	18	2	6	19644	153	53	5	27
parport.i.cil.c	35398	415	168	4	17	16656(*)	257(*)	97(*)	3(*)	26(*)
s3_clnt.blast.01.i.cil.c	35	4	47	11	47	132392	534	54	18	54
s3_clnt.blast.02.i.cil.c	38	5	56	14	56	354132	532	55	19	55
s3_clnt.blast.03.i.cil.c	38	5	46	11	46	196599	534	55	19	55
s3_clnt.blast.04.i.cil.c	39	5	72	18	72	172444	538	55	19	55
s3_srvr.blast.01.i.cil.c	98	4	86	21	86	452078	1142	98	26	97
s3_srvr.blast.02.i.cil.c	93	5	77	19	77	558114(*)	1185(*)	112(*)	32(*)	111(*)
s3_srvr.blast.03.i.cil.c	120	8	81	20	81	568769	1231	100	28	99
s3_srvr.blast.04.i.cil.c	106	6	80	20	80	–	–	–	–	–
s3_srvr.blast.06.i.cil.c	79	4	93	23	93	591029(*)	765(*)	73(*)	16(*)	72(*)
s3_srvr.blast.07.i.cil.c	100	6	84	21	84	517100(*)	769(*)	77(*)	21(*)	76(*)
s3_srvr.blast.08.i.cil.c	39	4	88	22	88	552926(*)	647(*)	57(*)	16(*)	57(*)
s3_srvr.blast.09.i.cil.c	207	5	80	20	80	540206(*)	846(*)	95(*)	21(*)	94(*)
s3_srvr.blast.10.i.cil.c	111	5	86	21	86	640117(*)	725(*)	68(*)	18(*)	67(*)
s3_srvr.blast.11.i.cil.c	94	5	68	17	68	496652(*)	945(*)	99(*)	25(*)	98(*)
s3_srvr.blast.12.i.cil.c	76	4	46	11	46	551382(*)	722(*)	67(*)	17(*)	66(*)
s3_srvr.blast.13.i.cil.c	100	6	80	20	80	482702(*)	1013(*)	104(*)	27(*)	103(*)
s3_srvr.blast.14.i.cil.c	100	6	91	22	91	561713(*)	721(*)	62(*)	15(*)	61(*)
s3_srvr.blast.15.i.cil.c	68	4	65	16	65	566797(*)	643(*)	62(*)	17(*)	62(*)
s3_srvr.blast.16.i.cil.c	94	5	99	24	99	705498(*)	734(*)	66(*)	17(*)	65(*)

Table 5.4: Comparison among different configurations of BLAST on safe programs. 'MO' indicates an "Out of Memory" error, and 'NP' a "No new predicates found during refinement" error.

Program	Blast 1 (-bfs -predH 0)	Blast 2 (-bfs -predH 7)	Blast 3 (-dfs -predH 0)	Blast 4 (-dfs -predH 7)	Blast B (best result)
test_locks_5.c	2.54	2.45	**1.39**	1.41	**1.39**
test_locks_6.c	5.58	5.17	**2.48**	2.58	**2.48**
test_locks_7.c	13.28	11.88	**4.41**	4.58	**4.41**
test_locks_8.c	29.75	28.63	**8.24**	8.48	**8.24**
test_locks_9.c	62.40	66.56	**16.07**	16.51	**16.07**
test_locks_10.c	187.27	179.19	**33.68**	34.29	**33.68**
test_locks_11.c	673.17	615.23	**75.93**	76.66	**75.93**
test_locks_12.c	>1800.00	>1800.00	**186.36**	187.86	**186.36**
test_locks_13.c	>1800.00	>1800.00	527.18	**523.81**	**523.81**
test_locks_14.c	>1800.00	>1800.00	1739.08	**1738.60**	**1738.60**
test_locks_15.c	>1800.00	>1800.00	>1800.00	>1800.00	>1800.00
cdaudio.i.cil.c	155.44	185.70	**69.87**	101.60	**69.87**
diskperf.i.cil.c	NP	>1800.00	NP	>1800.00	>1800.00
floppy.i.cil.c	**87.17**	>1800.00	NP	>1800.00	**87.17**
kbfiltr.i.cil.c	**8.68**	24.98	NP	11.85	**8.68**
parport.i.cil.c	**305.55**	844.14	NP	346.52	**305.55**
s3_clnt.blast.01.i.cil.c	30.01	238.98	**13.27**	516.54	**13.27**
s3_clnt.blast.02.i.cil.c	32.68	112.77	**25.56**	131.66	**25.56**
s3_clnt.blast.03.i.cil.c	52.87	192.29	**24.74**	131.63	**24.74**
s3_clnt.blast.04.i.cil.c	59.39	56.43	**26.54**	81.33	**26.54**
s3_srvr.blast.01.i.cil.c	522.66	MO	**388.27**	490.73	**388.27**
s3_srvr.blast.02.i.cil.c	MO	394.68	522.33	**158.00**	**158.00**
s3_srvr.blast.03.i.cil.c	585.94	186.59	445.72	**115.96**	**115.96**
s3_srvr.blast.04.i.cil.c	88.96	**72.45**	672.00	128.55	**72.45**
s3_srvr.blast.06.i.cil.c	MO	MO	302.82	**131.19**	**131.19**
s3_srvr.blast.07.i.cil.c	MO	MO	**214.89**	324.22	**214.89**
s3_srvr.blast.08.i.cil.c	MO	187.59	295.67	**46.08**	**46.08**
s3_srvr.blast.09.i.cil.c	MO	662.30	**196.07**	549.36	**196.07**
s3_srvr.blast.10.i.cil.c	MO	987.56	299.83	**46.24**	**46.24**
s3_srvr.blast.11.i.cil.c	896.28	519.18	**160.81**	440.06	**160.81**
s3_srvr.blast.12.i.cil.c	MO	598.52	299.56	**130.81**	**130.81**
s3_srvr.blast.13.i.cil.c	MO	MO	**256.79**	428.06	**256.79**
s3_srvr.blast.14.i.cil.c	MO	205.68	304.45	**131.26**	**131.26**
s3_srvr.blast.15.i.cil.c	MO	284.69	297.46	**46.34**	**46.34**
s3_srvr.blast.16.i.cil.c	MO	300.26	304.82	**130.88**	**130.88**
TOTAL (solved/time)	**19 / 3799.62**	**25 / 6963.90**	**30 / 7716.29**	**32 / 7213.65**	**33 / 5378.39**

successfully complete all benchmarks, and shows significant performance gains over SBE. SBE is able to solve only about one third of all benchmarks, and for the ones that complete, it is clearly outperformed by LBE. In Table 5.3, we see that SBE has in general a much larger ART. In Table 5.1 we observe not only that LBE performs significantly better than the

Table 5.5: Comparison among different configurations of Blast on programs with artificial bugs. 'MO' indicates an "Out of Memory" error, and 'NP' a "No new predicates found during refinement" error.

Program	Blast 1 (`-bfs -predH 0`)	Blast 2 (`-bfs -predH 7`)	Blast 3 (`-dfs -predH 0`)	Blast 4 (`-dfs -predH 7`)	Blast B (`best result`)
cdaudio.BUG.i.cil.c	43.01	36.74	9.84	**6.85**	**6.85**
diskperf.BUG.i.cil.c	**333.72**	>1800.00	344.95	>1800.00	**333.72**
floppy.BUG.i.cil.c	**44.45**	1035.71	47.18	1491.08	**44.45**
kbfiltr.BUG.i.cil.c	27.25	53.28	NP	**17.10**	**17.10**
parport.BUG.i.cil.c	2.23	4.01	**0.61**	0.82	**0.61**
s3_clnt.blast.01.BUG.i.cil.c	473.57	11.06	128.91	**3.18**	**3.18**
s3_clnt.blast.02.BUG.i.cil.c	49.69	**3.63**	53.51	4.75	**3.63**
s3_clnt.blast.03.BUG.i.cil.c	70.87	**2.64**	55.00	4.73	**2.64**
s3_clnt.blast.04.BUG.i.cil.c	79.13	**4.01**	57.90	4.54	**4.01**
s3_srvr.blast.01.BUG.i.cil.c	175.10	**3.79**	MO	52.42	**3.79**
s3_srvr.blast.02.BUG.i.cil.c	50.47	**2.94**	1054.91	75.74	**2.94**
s3_srvr.blast.03.BUG.i.cil.c	22.32	**3.04**	706.01	19.29	**3.04**
s3_srvr.blast.04.BUG.i.cil.c	32.07	**2.99**	1125.85	20.63	**2.99**
s3_srvr.blast.06.BUG.i.cil.c	814.84	22.24	243.14	**15.45**	**15.45**
s3_srvr.blast.07.BUG.i.cil.c	826.39	**133.34**	620.23	MO	**133.34**
s3_srvr.blast.08.BUG.i.cil.c	MO	29.56	229.42	**15.37**	**15.37**
s3_srvr.blast.09.BUG.i.cil.c	MO	**111.60**	597.43	MO	**111.60**
s3_srvr.blast.10.BUG.i.cil.c	MO	26.57	230.60	**15.29**	**15.29**
s3_srvr.blast.11.BUG.i.cil.c	346.27	**19.21**	717.02	85.91	**19.21**
s3_srvr.blast.12.BUG.i.cil.c	275.10	**15.13**	242.42	15.25	**15.13**
s3_srvr.blast.13.BUG.i.cil.c	386.99	**105.12**	741.39	301.39	**105.12**
s3_srvr.blast.14.BUG.i.cil.c	350.47	21.25	242.29	**15.33**	**15.33**
s3_srvr.blast.15.BUG.i.cil.c	MO	31.26	231.16	**15.39**	**15.39**
s3_srvr.blast.16.BUG.i.cil.c	460.34	21.95	242.94	**15.47**	**15.47**
TOTAL (solved/time)	**20 / 4864.28**	**23 / 1701.07**	**22 / 7922.71**	**21 / 2195.98**	**24 / 905.65**

`-dfs -predH 7` configuration of Blast, but that LBE is better than any Blast configuration (column `best result`). LBE performed best also in finding the error paths (see Table 5.2), clearly outperforming both SBE and Blast.

In summary, the experiments show that the LBE approach outperforms the SBE approach, both for correct and defective programs. This provides evidence of the benefits of a "more symbolic" analysis as performed in the LBE approach.

One might argue that our CPAchecker-based SBE implementation might be sub-optimal, although it uses the same implementation and execution environment as LBE; in fact, both implementations currently suffer

Table 5.6: Comparison between using MathSAT and CSIsat as interpolation procedures in Blast. 'MO' indicates an "Out of Memory" error, 'NP' a "No new predicates found during refinement" error, and 'SAT' indicates that CSIsat incorrectly reported a satisfiable result for an unsatisfiable query.

Safe programs (using options `-dfs -predH 7`)

Program	Blast + MathSAT	Blast + CSIsat
test_locks_5.c	1.41	**1.01**
test_locks_6.c	2.58	**1.97**
test_locks_7.c	4.58	**3.68**
test_locks_8.c	8.48	**7.23**
test_locks_9.c	16.51	**14.91**
test_locks_10.c	34.29	**32.50**
test_locks_11.c	76.66	**74.18**
test_locks_12.c	187.86	**183.45**
test_locks_13.c	**523.81**	526.50
test_locks_14.c	**1738.60**	1785.70
test_locks_15.c	>1800.00	>1800.00
cdaudio.i.cil.c	**101.60**	NP
diskperf.i.cil.c	>1800.00	>1800.00
floppy.i.cil.c	>1800.00	>1800.00
kbfiltr.i.cil.c	**11.85**	28.87
parport.i.cil.c	**346.52**	NP
s3_clnt.blast.01.i.cil.c	**516.54**	NP
s3_clnt.blast.02.i.cil.c	**131.66**	NP
s3_clnt.blast.03.i.cil.c	**131.63**	NP
s3_clnt.blast.04.i.cil.c	**81.33**	NP
s3_srvr.blast.01.i.cil.c	**490.73**	NP
s3_srvr.blast.02.i.cil.c	**158.00**	SAT
s3_srvr.blast.03.i.cil.c	**115.96**	NP
s3_srvr.blast.04.i.cil.c	**128.55**	NP
s3_srvr.blast.06.i.cil.c	**131.19**	MO
s3_srvr.blast.07.i.cil.c	**324.22**	MO
s3_srvr.blast.08.i.cil.c	**46.08**	SAT
s3_srvr.blast.09.i.cil.c	**549.36**	SAT
s3_srvr.blast.10.i.cil.c	**46.24**	NP
s3_srvr.blast.11.i.cil.c	**440.06**	MO
s3_srvr.blast.12.i.cil.c	**130.81**	SAT
s3_srvr.blast.13.i.cil.c	**428.06**	MO
s3_srvr.blast.14.i.cil.c	**131.26**	MO
s3_srvr.blast.15.i.cil.c	**46.34**	MO
s3_srvr.blast.16.i.cil.c	**130.88**	SAT
TOTAL (solved/time)	**32 / 7213.65**	**11 / 2660.00**

Unsafe programs (using options `-bfs -predH 7`)

Program	Blast + MathSAT	Blast + CSIsat
cdaudio.BUG.i.cil.c	36.74	**33.40**
diskperf.BUG.i.cil.c	>1800.00	NP
floppy.BUG.i.cil.c	**1035.71**	>1800.00
kbfiltr.BUG.i.cil.c	**53.28**	NP
parport.BUG.i.cil.c	**4.01**	8.17
s3_clnt.blast.01.BUG.i.cil.c	**11.06**	17.85
s3_clnt.blast.02.BUG.i.cil.c	**3.63**	6.05
s3_clnt.blast.03.BUG.i.cil.c	**2.64**	9.34
s3_clnt.blast.04.BUG.i.cil.c	**4.01**	7.74
s3_srvr.blast.01.BUG.i.cil.c	**3.79**	SAT
s3_srvr.blast.02.BUG.i.cil.c	**2.94**	4.73
s3_srvr.blast.03.BUG.i.cil.c	**3.04**	3.50
s3_srvr.blast.04.BUG.i.cil.c	**2.99**	3.61
s3_srvr.blast.06.BUG.i.cil.c	22.24	**19.59**
s3_srvr.blast.07.BUG.i.cil.c	**133.34**	SAT
s3_srvr.blast.08.BUG.i.cil.c	**29.56**	42.19
s3_srvr.blast.09.BUG.i.cil.c	**111.60**	SAT
s3_srvr.blast.10.BUG.i.cil.c	**26.57**	52.58
s3_srvr.blast.11.BUG.i.cil.c	19.21	**15.93**
s3_srvr.blast.12.BUG.i.cil.c	**15.13**	16.17
s3_srvr.blast.13.BUG.i.cil.c	**105.12**	SAT
s3_srvr.blast.14.BUG.i.cil.c	21.25	**18.46**
s3_srvr.blast.15.BUG.i.cil.c	**31.26**	44.87
s3_srvr.blast.16.BUG.i.cil.c	21.95	**18.59**
TOTAL (solved/time)	**23 / 1701.07**	**17 / 322.77**

from some inefficiencies and have room for several optimizations. Therefore, we compare also with Blast. By looking at Tables 5.1 and 5.2, we see that LBE outperforms also Blast, despite the fact that the latter is

the result of several years of fine-tuning. BLAST in turn is much more efficient than SBE. However, the performance gap between BLAST and SBE highly depends on the command-line options used for BLAST: the performance of BLAST varies significantly with different command-line options, as demonstrated by the results reported in Tables 5.4 and 5.5, where the four different configurations of BLAST that we have tried are compared.

Finally, it is also important to observe that in all experiments with BLAST we have used MATHSAT as interpolation procedure, instead of the default one CSISAT [BZM08]. The reason for this is not only that MATHSAT is generally faster than CSISAT (see §4.6), but also – and more importantly – that we wanted to minimize the differences between BLAST and CPACHECKER in terms of the kind of predicates that are automatically discovered during abstraction refinement, since this is an extremely important factor for performance of CEGAR-based approaches. In fact, when we tried to run BLAST with CSISAT as interpolation procedure, the performance was significantly worse, as shown by the results reported in Table 5.6. [5] We also tried to use MATHSAT instead of SIMPLIFY for computing abstract successor states with SBE, but this did not improve performance. On the contrary, BLAST performed worse in this case. The reason is that with SBE the SMT solver gets invoked very frequently and with very small formulae, and MATHSAT is not optimized for this scenario.

5.4.4 Comparison with SatAbs

In the last part of our experiments, we compare our LBE implementation with the SATABS tool [CKSY04]. The comparison is interesting in principle because the approach followed by SATABS bears similarities with both LBE and SBE. On the one hand, SATABS uses a fully symbolic CEGAR approach – by first abstracting the whole program into a Boolean program,

[5] We ran this comparison only for the best configurations of BLAST.

Table 5.7: Comparison between CPAchecker-LBE and SatAbs on simplified benchmark instances. For SatAbs, an 'RF' entry indicates a "Refinement failure" error.

Safe programs

Program	CPAchecker-LBE	SatAbs
test_locks_5.c	**0.10**	0.72
test_locks_6.c	**0.11**	1.11
test_locks_7.c	**0.12**	1.69
test_locks_8.c	**0.13**	2.41
test_locks_9.c	**0.14**	3.72
test_locks_10.c	**0.15**	5.43
test_locks_11.c	**0.15**	7.67
test_locks_12.c	**0.15**	10.32
test_locks_13.c	**0.16**	15.17
test_locks_14.c	**0.17**	21.07
test_locks_15.c	**0.18**	22.84
cdaudio_SIMPL	**11.22**	RF
diskperf_SIMPL	**54.09**	RF
floppy_SIMPL	**8.96**	RF
kbfiltr_SIMPL	**1.69**	26.51
s3_clnt_1_SIMPL	**11.84**	1002.18
s3_clnt_2_SIMPL	**4.15**	>1800.00
s3_clnt_3_SIMPL	**5.99**	>1800.00
s3_clnt_4_SIMPL	**10.34**	1475.30
s3_srvr_1_SIMPL	**135.30**	1493.43
s3_srvr_2_SIMPL	**152.02**	843.55
s3_srvr_3_SIMPL	**65.06**	939.82
s3_srvr_4_SIMPL	**188.17**	748.60
TOTAL (solved/time)	**23 / 650.39**	**18 / 6621.54**
TOTAL w/o `test_locks*`	**12 / 648.83**	**7 / 6529.39**

Unsafe programs

Program	CPAchecker-LBE	SatAbs
cdaudio_SIMPL_BUG	**5.92**	RF
diskperf_SIMPL_BUG	**2.03**	12.84
floppy_SIMPL_BUG	**5.40**	RF
kbfiltr_SIMPL_BUG	**1.27**	RF
s3_clnt_1_SIMPL_BUG	**1.51**	44.22
s3_clnt_2_SIMPL_BUG	**1.24**	46.70
s3_clnt_3_SIMPL_BUG	**1.52**	47.78
s3_clnt_4_SIMPL_BUG	**1.73**	47.40
s3_srvr_1_SIMPL_BUG	**0.70**	77.51
s3_srvr_2_SIMPL_BUG	**0.68**	77.33
s3_srvr_3_SIMPL_BUG	**0.71**	76.45
s3_srvr_4_SIMPL_BUG	**0.66**	76.53
TOTAL (solved/time)	**12 / 23.37**	**9 / 506.76**

and then using standard symbolic model checking techniques for analyzing the abstract program – which in some sense can be thought of as pushing to the extreme the idea behind LBE of reducing the amount of explicit search in favor of more symbolic techniques. On the other hand, instead, in order to construct the abstract program, each basic block of the input program is abstracted separately, similarly to what is done with SBE.

From the practical point of view, however, the comparison presents several difficulties. First, SatAbs uses a bit-accurate representation of data types and constructs of C programs, whereas in our LBE implementation within CPAchecker we model program variables using unbounded integers, as done in Blast. Second, in our current implementation of LBE we

do not take the semantics of pointers into account (treating them as regular variables), and we treat arrays as uninterpreted functions, whereas in SATABS both are modeled precisely. Therefore, SATABS is potentially much more precise than our current implementation of LBE. [6] However, this increased precision could also have a non-negligible computational cost. Since all the benchmarks that we have collected do not require such increased precision, therefore, the comparison on them would be biased in favor of LBE.

In order to mitigate (at least partially) the effects of such differences, we have created some simplified benchmark instances, obtained from a subset of the programs that we used in the comparison with BLAST by manually replacing all pointer dereferences and accesses to fields of data structures with fresh variables and by removing bit-level operations. The results of the comparison between LBE and SATABS on these instances are reported in Table 5.7. (For the experiments, we used SATABS version 2.4 with CADENCE-SMV as model checker.) From Table 5.7 we can see that LBE is significantly faster than SATABS on such instances, with gaps of an order of magnitude on average. [7] Moreover, in several cases SATABS fails to complete the analysis, exiting with a "Refinement failure" error (indicated with 'RF'). We thought that the significant performance gap could be attributed, at least in part, to the fact that SATABS models all variables as bit-vectors in the abstraction and refinement phases. However,

[6] We remark that this is not because of some intrinsic limitations of LBE or of CPACHECKER, but only because the current implementation is still a prototype.

[7] In principle, the "manual" simplifications that we performed to obtain the simplified instances are very similar to what is done internally by CPACHECKER. Therefore, one could expect the execution times of CPACHECKER-LBE on such instances to be substantially identical to the corresponding ones on the original instances. In some cases, however, we observe significant gaps between the "normal" and the "simplified" version of a program. This can be explained by observing that the formulae given to MATHSAT in the two cases might be slightly different, and this can lead to differences in the search space of the DPLL engine. It is a well-known fact in the SAT and SMT communities that even minor syntactical variations in the input problem can have a very big impact on the performance of DPLL.

by analyzing the output produced by SatAbs, we found that most of the time (about 95% for safe programs, and about 85% for unsafe ones) is spent in model checking the abstract Boolean program, and not in computing the abstraction and refining it. Therefore, we think that this shows that the lazy abstraction approach with LBE works better than a fully symbolic CEGAR for these benchmarks.

Chapter 6

Conclusions

Formal methods are becoming increasingly important for debugging and verifying hardware and software systems, whose current complexity makes the traditional approaches based on testing increasingly-less adequate.

One of the most promising research directions in formal verification is based on the exploitation of Satisfiability Modulo Theories (SMT), an emerging paradigm for checking the satisfiability of logical formulae expressed in a combination of decidable first-order theories. SMT solvers have seen tremendous improvements over the last few years, and they are now able to deliver the same high levels of automation, efficiency and scalability of propositional SAT solvers – which are at the basis of a number of successful verification techniques – while at the same time offering a much higher expressive power.

In order to fully exploit the potential of SMT in formal verification, SMT solvers should provide functionalities that go beyond simply checking the satisfiability of a formula, such as model generation and enumeration, proof-production, extraction of unsatisfiable cores and computation of interpolants.

In this thesis, we have presented MathSAT, a modern, efficient SMT solver that provides several important functionalities, and can be used

as a workhorse engine in formal verification. We have developed novel algorithms for the extraction of unsatisfiable cores and the generation of interpolants in SMT that significantly advance the state of the art, taking full advantage of modern SMT techniques. In order to demonstrate the usefulness and potential of SMT in verification, we have developed a novel technique for software model checking, that fully exploits the power and functionalities of the SMT engine, showing that this leads to significant improvements in performance.

The work in this thesis opens a number of future research directions to explore. First, MathSAT can be further developed to support other important theories, and to improve the functionalities that it provides. In particular, a natural research direction worth investigating is the possibility of extending the generation of interpolants to other important theories such as linear integer arithmetic, some fragments of the theory of arrays, and the theory of bit-vectors. Other possibilities are the investigation of techniques for quantifier elimination and for simplifications of formulae, and the improvement of the proof-production capabilities of MathSAT. Finally, techniques for better exploiting SMT solvers in formal verification can be further investigated, considering in particular approaches based on Bounded Model Checking, interpolation, and abstraction-refinement, exploiting the integration of MathSAT within the NuSMV symbolic model checker that is already ongoing.

Bibliography

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In A. Voronkov, editor, *Proceedings of CADE-18*, volume 2392 of *LNCS*, pages 195–210. Springer, 2002.

[ABM07] A. Armando, M. Benerecetti, and J. Mantovani. Abstraction Refinement of Linear Programs with Arrays. In O. Grumberg and M. Huth, editors, *Proceedings of TACAS'07*, volume 4424 of *LNCS*, pages 373–388. Springer, 2007.

[Ack54] W. Ackermann. *Solvable Cases of the Decision Problem*. North Holland Pub. Co., 1954.

[AMP09] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Int. J. Softw. Tools Technol. Transf.*, 11(1):69–83, 2009.

[ANORC08] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez Carbonell. Efficient Generation of Unsatisfiability Proofs and Cores in SAT. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of LPAR'08*, volume 5330 of *LNCS*, pages 16–30. Springer, 2008.

[BB04] C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In R. Alur and D. A. Peled, editors, *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 515–518. Springer, 2004.

[BB09a] R. Brummaryer and A. Biere. Lemmas on Demand for the Extensional Theory of Arrays. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6, 2009.

[BB09b] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In S. Kowalewski and A. Philippou, editors, *Proceedings of TACAS'09*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.

[BBC+05] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. MathSAT: Tight Integration of SAT and Mathematical Decision Procedures. *Journal of Automated Reasoning*, 35(1-3):265–293, 2005.

[BBC+06a] M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzén, Z. Hanna, Z. Khasidashvili, A. Palti, and R. Sebastiani. Encoding RTL Constructs for MathSAT: a Preliminary Report. *Electr. Notes Theor. Comput. Sci.*, 144(2):3–14, 2006.

[BBC+06b] M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient Theory Combination via Boolean Search. *Inf. Comput.*, 204(10):1493–1525, 2006.

[BCF+07] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification

Problems. In W. Damm and H. Hermanns, editors, *Proceedings of CAV'07*, volume 4590 of *LNCS*. Springer, 2007.

[BCF+08] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: A Comparative Analysis. Extended version. *Annals of Mathematics and Artificial Intelligence.*, 2008. To appear.

[BCG+09] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software Model Checking via Large-Block Encoding. In *Proceedings of FMCAD'09*, 2009. To appear. Technical report available at `http://arxiv.org/abs/0904.4709`.

[BCLZ04] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement. In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 457–461. Springer, 2004.

[BD02] Raik Brinkmann and Rolf Drechsler. Rtl-datapath verification using integer linear programming. In *Proceedings of ASP-DAC'02*, page 741, Washington, DC, USA, 2002. IEEE Computer Society.

[BDL98] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of DAC*, pages 522–527, 1998.

[BDS02] C. W. Barrett, D. L. Dill, and A. Stump. A Generalization of Shostak's Method for Combining Decision Procedures. In *Proceedings of FroCos'02*, volume 2309 of *LNCS*, pages 132–146. Springer, 2002.

[BH07] D. Babic and A. J. Hu. Structural Abstraction of Software Verification Conditions. In W. Damm and H. Hermanns, editors, *Proceedings of CAV'07*, volume 4590 of *LNCS*, pages 366–378. Springer, 2007.

[BH08] D. Babic and A. J. Hu. CALYSTO: scalable and precise extended static checking. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *Proceedings of ICSE'08*, pages 211–220. ACM, 2008.

[BHJM07] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *STTT*, 9(5-6):505–525, 2007.

[Bie08a] A. Biere. Adaptive restart strategies for conflict driven sat solvers. In H. K. Büning and X. Zhao, editors, *Proceedings of SAT'08*, volume 4996 of *LNCS*, pages 28–33. Springer, 2008.

[Bie08b] A. Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, May 2008.

[BK09] D. Beyer and M. E. Keremoglu. CPACHECKER: A Tool for Configurable Software Verification. Technical Report SFU-CS-2009-002, Simon Fraser University, January 2009.

[BKO+09] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. An abstraction-based decision procedure for bit-vector arithmetic. *Int. J. Softw. Tools Technol. Transf.*, 11(2):95–104, 2009.

[BL05] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In M. D. Ernst and T. P. Jensen, editors, *Proceedings of PASTE'05*, pages 82–87. ACM, 2005.

[BLM05] T. Ball, S. K. Lahiri, and M. Musuvathi. Zap: Automated theorem proving for software analysis. In G. Sutcliffe and A. Voronkov, editors, *Proceedings of LPAR'05*, volume 3835 of *LNCS*, pages 2–22. Springer, 2005.

[BMS06] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *Proceedigs of VMCAI'06*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[BNO+08a] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez Carbonell, and A. Rubio. The Barcelogic SMT Solver. In A. Gupta and S. Malik, editors, *Proceedings of CAV'08*, volume 5123 of *LNCS*, pages 294–298. Springer, 2008.

[BNO+08b] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. A Write-Based Solver for SAT Modulo the Theory of Arrays. In A. Cimatti and R. B. Jones, editors, *Proceedings of FMCAD'08*, pages 101–108. IEEE, 2008.

[BNOT06] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on Demand in SAT Modulo Theories. In M. Hermann and A. Voronkov, editors, *Proceedings of LPAR'06*, volume 4246 of *LNCS*, pages 512–526. Springer, 2006.

[BPR03] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking c programs. *STTT*, 5(1):49–58, 2003.

[BPST09] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. OpenSMT 0.2 System Description. Available at `http://verify.inf.unisi.ch/opensmt`, August 2009.

[BR02] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of POPL*, pages 1–3, 2002.

[Bru09] R. Bruttomesso. An Extension of the Davis-Putnam Procedure and its Application to Preprocessing in SMT. In *Proceedings of SMT'09*, Montreal, Canada, 2009.

[BSST09] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[BT07] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of CAV'07*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[BZM08] D. Beyer, D. Zufferey, and R. Majumdar. CSISAT: Interpolation for LA+EUF. In A. Gupta and S. Malik, editors, *Proceedings of CAV'08*, volume 5123 of *LNCS*, pages 304–308. Springer, 2008.

[CCF+07] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar. Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In *Proceedings of FMCAD'07*, pages 69–76. IEEE Computer Society, 2007.

[CCG+02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *Proceedings*

of CAV'02, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGS07] A. Cimatti, A. Griggio, and R. Sebastiani. A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories. In J. Marques-Silva and K. A. Sakallah, editors, *Proceedings of SAT'07*, volume 4501 of *LNCS*, pages 334–339. Springer, 2007.

[CGS08] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of TACAS'08*, volume 4963 of *LNCS*, pages 397–412. Springer, 2008.

[CGS09a] A. Cimatti, A. Griggio, and R. Sebastiani. Computing Small Unsatisfiable Cores in Satisfiability Modulo Theories. Submitted for publication, 2009.

[CGS09b] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. *CoRR*, abs/0906.4492, 2009. Submitted for publication.

[CGS09c] A. Cimatti, A. Griggio, and R. Sebastiani. Interpolant Generation for UTVPI. In R. A. Schmidt, editor, *Proceedings of CADE-22*, volume 5663 of *LNCS*, pages 167–182. Springer, 2009.

[CKL04] E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[CKSY04] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate Abstraction of ANSI-C Programs Using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[CM06] S. Cotton and O. Maler. Fast and Flexible Difference Constraint Propagation for DPLL(T). In A. Biere and C. P. Gomes, editors, *Proceedings of SAT'06*, volume 4121 of *LNCS*, pages 170–183. Springer, 2006.

[CMNQ06] G. Cabodi, M. Murciano, S. Nocco, and S. Quer. Stepping forward with interpolants in unbounded model checking. In S. Hassoun, editor, *Proceedings of ICCAD'06*, pages 772–778. ACM, 2006.

[DDA09] I. Dillig, T. Dillig, and A. Aiken. Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In A. Bouajjani and O. Maler, editors, *Proceedings of CAV'09*, volume 5643 of *LNCS*. Springer, 2009.

[DdM06a] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In T. Ball and R. B. Jones, editors, *Proceedings of CAV'06*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[DdM06b] B. Dutertre and L. de Moura. Integrating Simplex with DPLL(T). Technical Report CSL-06-01, SRI, 2006.

[DdM06c] B. Dutertre and L. de Moura. System Description: Yices 1.0. Available at `http://yices.csl.sri.com/yices-smtcomp06.pdf`, 2006.

[DHN06] N. Dershowitz, Z. Hanna, and A. Nadel. A Scalable Algorithm for Minimal Unsatisfiable Core Extraction. In A. Biere and C. P. Gomes, editors, *Proceedings of SAT'06*, volume 4121 of *LNCS*, pages 36–41. Springer, 2006.

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[dMB08a] L. de Moura and N. Bjørner. Model-based theory combination. *Electron. Notes Theor. Comput. Sci.*, 198(2):37–49, 2008.

[dMB08b] L. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In P. Rudnicki, G. Sutcliffe, B. Konev, R. A. Schmidt, and S. Schulz, editors, *Proceedings of the LPAR'08 Workshops*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[dMB08c] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[dMOR+04] L. de Moura, S. Owre, H. Ruess, J. Rushby, and N. Shankar. The ICS Decision Procedures for Embedded Deduction. In *Proceedings of IJCAR'04*, volume 3097 of *LNCS*, pages 218–222. Springer, 2004.

[dMRS02] L. de Moura, H. Rueß, and M. Sorea. Lemmas on demand for satisfiability solvers. In *Proceedings of SAT'02*, LNCS. Springer, 2002.

[DNS05] D. Detlefs, G. Nelson, and J. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[EKS06] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with craig interpolation and symbolic pushdown systems. In H. Hermanns and J. Palsberg, editors, *Proceedings of TACAS'06*, volume 3920 of *LNCS*, pages 489–503. Springer, 2006.

[End01] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2^{nd} edition, 2001.

[FGG+09] A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli. Ground interpolation for the theory of equality. In S. Kowalewski and A. Philippou, editors, *Proceedings of TACAS'09*, volume 5505 of *LNCS*, pages 413–427. Springer, 2009.

[FJOS03] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem Proving Using Lazy Proof Explication. In *Proceedings of CAV'03*, LNCS. Springer, 2003.

[FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. Ics: Integrated canonizer and solver. In *Proceedings of CAV'01*, LNCS, pages 246–249. Springer, 2001.

[GD07] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In W. Damm and H. Hermanns, editors,

Proceedings of CAV'07, volume 4590 of *LNCS*, pages 519–531, 2007.

[GKF08] A. Goel, S. Krstić, and A. Fuchs. Deciding array formulas with frugal axiom instantiation. In *Proceedings of SMT'08/BPR'08*, pages 12–17, New York, NY, USA, 2008. ACM.

[GKS08] R. Gershman, M. Koifman, and O. Strichman. An approach for extracting a small unsatisfiable core. *Formal Methods in System Design*, 33(1-3):1–27, 2008.

[GKT09] A. Goel, S. Krstic, and C. Tinelli. Ground Interpolation for Combined Theories. In R. A. Schmidt, editor, *Proceedings of CADE-22*, volume 5663 of *LNCS*, pages 183–198. Springer, 2009.

[GLST05] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. *SIGPLAN Not.*, 40(1):122–131, 2005.

[GMP] The GNU Multiple Precision Arithmetic Library (GMP). `http://gmplib.org`.

[GNRZ07] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Ann. Math. Artif. Intell.*, 50(3-4):231–254, 2007.

[GSZ+98] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD Based Procedures for a Theory of Equality with Uninterpreted Functions. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of CAV'98*, volume 1427 of *LNCS*, pages 244–255. Springer, 1998.

[HJMM04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In N. D. Jones and X. Leroy, editors, *Proceedings of POPL'04*, pages 232–244. ACM, 2004.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of POPL*, pages 58–70, 2002.

[HS97] W. Harvey and P. Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming. In *Australian Computer Science Conference (Australian Computer Science Communications)*, pages 102–111, 1997.

[Hua05] J. Huang. MUP: a minimal unsatisfiability prover. In *Proceedings of ASP-DAC'05*, pages 432–437, New York, NY, USA, 2005. ACM Press.

[JCG08] H. Jain, E. M. Clarke, and O. Grumberg. Efficient Craig Interpolation for Linear Diophantine (Dis)Equations and Linear Modular Equations. In A. Gupta and S. Malik, editors, *Proceedings of CAV'08*, volume 5123 of *LNCS*, pages 254–267. Springer, 2008.

[JLS09] S. Jha, R. Limaye, and S. A. Seshia. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In A. Bouajjani and O. Maler, editors, *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 668–674. Springer, 2009.

[JM05] R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In K. Etessami and S. K. Rajamani, editors, *Proceedings of CAV'05*, volume 3576 of *LNCS*, pages 39–51. Springer, 2005.

[JM06] R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In H. Hermanns and J. Palsberg, editors, *Proceedings of TACAS'06*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

[JM07] R. Jhala and K. L. McMillan. Array Abstractions from Proofs. In W. Damm and H. Hermanns, editors, *Proceedings of CAV'07*, volume 4590 of *LNCS*, pages 193–206. Springer, 2007.

[JMSY94] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap. Beyond Finite Domains. In *Proceedings of PPCP*, volume 874 of *LNCS*, pages 86–94. Springer, 1994.

[JMX07] R. Jhala, R. Majumdar, and R. Xu. State of the Union: Type Inference Via Craig Interpolation. In O. Grumberg and M. Huth, editors, *Proceedings of TACAS'07*, volume 4424 of *LNCS*, pages 553–567. Springer, 2007.

[JS05] P. Jackson and D. Sheridan. Clause Form Conversions for Boolean Circuits. In H. H. Hoos and D. G. Mitchell, editors, *Proceedings of SAT'04*, volume 3542 of *LNCS*, pages 183–198. Springer, 2005.

[KG07] S. Krstic and A. Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *Proceedings of FroCoS'07*, volume 4720 of *LNAI*, pages 1–27. Springer, 2007.

[Kha79] L. G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[KMZ06] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In M. Young and P. T. Devanbu, editors, *Proceedings of FSE'05*, pages 105–116. ACM, 2006.

[KSJ09] H. Kim, F. Somenzi, and H. Jin. Efficient Term-ITE Conversion for Satisfiability Modulo Theories. In O. Kullmann, editor, *Proceedings of SAT'09*, volume 5584 of *LNCS*, pages 195–208. Springer, 2009.

[KW07] D. Kroening and G. Weissenbacher. Lifting Propositional Interpolants to the Word-Level. In *Proceedings of FMCAD'07*, pages 85–89, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[KZ05] D. Kapur and C. Zarba. A reduction approach to decision procedures. Technical Report TR-CS-1005-44, University of New Mexico, 2005.

[LM05] S. K. Lahiri and M. Musuvathi. An Efficient Decision Procedure for UTVPI Constraints. In B. Gramlich, editor, *Proceedings of FroCos'05*, volume 3717 of *LNCS*, pages 168–183. Springer, 2005.

[LMS04] I. Lynce and J. P. Marques-Silva. On Computing Minimum Unsatisfiable Cores. In *Proceedings of SAT'04*, LNCS. Springer, 2004.

[LNO06] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In T. Ball and R. B. Jones, editors, *Proceedings of CAV'06*, volume 4144 of *LNCS*, pages 413–426. Springer, 2006.

[LQ08] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In G. C. Necula and P. Wadler, editors, *Proceedings of POPL'08*, pages 171–182. ACM, 2008.

[LQR09] S. K. Lahiri, S. Qadeer, and Z. Rakamaric. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In A. Bouajjani and O. Maler, editors, *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 509–524. Springer, 2009.

[LS06] B. Li and F. Somenzi. Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking. In H. Hermanns and J. Palsberg, editors, *Proceedings of TACAS'06*, volume 3920 of *LNCS*, pages 227–241. Springer, 2006.

[MA03] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *Proceedings of TACAS'03*, volume 2619 of *LNCS*, pages 2–17. Springer, 2003.

[McM92] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA, 1992.

[McM02] K. L. McMillan. Applying sat methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 250–264. Springer, 2002.

[McM03] K. L. McMillan. Interpolation and SAT-Based Model Checking. In W. A. Hunt Jr. and F. Somenzi, editors, *Proceedings of CAV'03*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[McM05] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[McM06] K. L. McMillan. Lazy Abstraction with Interpolants. In T. Ball and R. B. Jones, editors, *Proceedings of CAV'06*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[McM08] K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of TACAS'08.*, volume 4963 of *LNCS*, pages 413–427. Springer, 2008.

[Min01] A. Miné. The Octagon Abstract Domain. In *Proceedings of WCRE*, pages 310–, 2001.

[MLA+05] M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. Marques-Silva, and K. A. Sakallah. A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas. In F. Bacchus and T. Walsh, editors, *Proceedings of SAT'05*, volume 3569 of *LNCS*, pages 467–474. Springer, 2005.

[Mon09] D. Monniaux. On using floating-point computations to help an exact linear arithmetic decision procedure. In A. Bouajjani and O. Maler, editors, *Proceedings of CAV'09*, volume 5643 of *LNCS*. Springer, 2009.

[MS07] J. Marques-Silva. Interpolant Learning and Reuse in SAT-Based Model Checking. *Electr. Notes Theor. Comput. Sci.*, 174(3):31–43, 2007.

[MSV07] P. Manolios, S. K. Srinivasan, and D. Vroon. BAT: The Bit-Level Analysis Tool. In W. Damm and H. Hermanns, editors,

Proceedings of CAV'07, volume 4590 of *LNCS*, pages 303–306. Springer, 2007.

[MV07] P. Manolios and D. Vroon. Efficient Circuit to CNF Conversion. In J. Marques-Silva and K. A. Sakallah, editors, *Proceedings of SAT'07*, volume 4501 of *LNCS*, pages 4–9. Springer, 2007.

[MZ03] Z. Manna and C. G. Zarba. Combining Decision Procedures. In *In Formal Methods at the Cross Roads: From Panacea to Foundational Support*, LNCS, pages 381–422. Springer, 2003.

[NMRW02] G. C. Necula, S. McPeak, S. Prakash Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In R. N. Horspool, editor, *Proceedings of CC'02*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[NO79] C. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2):245–257, 1979.

[NO05] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic. In *Proceedings of CAV'05*, volume 3576 of *LNCS*, pages 321–334. Springer, 2005.

[NO07] R. Nieuwenhuis and A. Oliveras. Fast Congruence Closure and Extensions. *Inf. Comput.*, 2005(4):557–580, 2007.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[OMA+04] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: A Minimally-Unsatisfiable Subformula Extractor. In *Proceedings of DAC'04*, pages 518–523. ACM/IEEE, 2004.

[Opp80] D. C. Oppen. Complexity, Convexity and Combinations of Theories. *Theoretical Computer Science*, 12:291–302, 1980.

[Pap81] C. H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981.

[Pud97] P. Pudlák. Lower bounds for resolution and cutting planes proofs and monotone computations. *J. of Symb. Logic*, 62(3), 1997.

[Pug91] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of SC*, pages 4–13, 1991.

[RS04] H. Rueß and N. Shankar. Solving linear arithmetic constraints. Technical Report CSL-SRI-04-01, SRI International, Computer Science Laboratory, January 2004.

[RSS07] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In B. Cook and A. Podelski, editors, *Proceedings of VMCAI'07*, volume 4349 of *LNCS*, pages 346–362. Springer, 2007.

[RT06] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smt-lib.org`, 2006.

[SBDL01] A. Stump, C. W. Barrett, D. L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Proceed-*

ings of LICS'01, page 29, Washington, DC, USA, 2001. IEEE Computer Society.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

[SDPK09] C. Scholl, S. Disch, F. Pigorsch, and S. Kupferschmid. Computing Optimized Representations for Non-convex Polyhedra by Detection and Removal of Redundant Linear Constraints. In S. Kowalewski and A. Philippou, editors, *Proceedings of TACAS'09*, volume 5505 of *LNCS*, pages 383–397. Springer, 2009.

[Seb07] R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 3(3-4):141–224, 2007.

[SFBD08] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler. Using unsatisfiable cores to debug multiple design errors. In *Proceedings of GLSVLSI'08*, pages 77–82, New York, NY, USA, 2008. ACM.

[Sho79] R. Shostak. A Pratical Decision Procedure for Arithmetic with Function Symbols. *Journal of the ACM*, 26(2):351–360, 1979.

[Sho84] R.E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31:1–12, 1984.

[SI09] C. Sinz and M. Iser. Problem-sensitive restart heuristics for the dpll procedure. In O. Kullmann, editor, *Proceedings of SAT'09*, volume 5584 of *LNCS*, pages 356–362. Springer, 2009.

[SMT] The SMT solvers competition (SMT-COMP). `http://smtcomp.org`.

[SR02] N. Shankar and H. Rueß. Combining shostak theories. In *Proceedings of RTA'02*, pages 1–18. Springer, 2002.

[SS08] V. Sofronie-Stokkermans. Interpolation in Local Theory Extensions. *Logical Methods in Computer Science (Special issue dedicated to IJCAR 2006)*, 4(4):Paper 1, 2008.

[SSB02] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding Separation Formulas with SAT. In E. Brinksma and K. G. Larsen, editors, *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 209–222. Springer, 2002.

[Tse68] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125, 1968.

[Van01] R. J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, 2001.

[vG07] A. van Gelder. Verifying Propositional Unsatisfiability: Pitfalls to Avoid. In J. Marques-Silva and K. A. Sakallah, editors, *Proceedings of SAT'07*, volume 4501 of *LNCS*, pages 328–333. Springer, 2007.

[vMW08] H. van Maaren and S. Wieringa. Finding guaranteed muses fast. In H. K. Büning and X. Zhao, editors, *Proceedings of SAT'08*, volume 4996 of *LNCS*, pages 291–304. Springer, 2008.

[WFG+07] R. Wille, G. Fey, D. Große, S. Eggersglüß, and Rolf Drechsler. SWORD: A SAT like prover using word level information. In *Proceedings of IFIP VLSI-SoC'07*, pages 88–93. IEEE, 2007.

[WKG07] C. Wang, H. Kim, and A. Gupta. Hybrid cegar: combining variable hiding and predicate abstraction. In *Proceedings of*

ICCAD'07, pages 310–317, Piscataway, NJ, USA, 2007. IEEE Press.

[YM05] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In R. Nieuwenhuis, editor, *Proceedings of CADE-20*, volume 3632 of *LNCS*, pages 353–368. Springer, 2005.

[YM06] Y. Yu and S. Malik. Lemma Learning in SMT on Linear Constraints. In A. Biere and C. P. Gomes, editors, *Proceedings of SAT'06*, volume 4121 of *LNCS*, pages 142–155. Springer, 2006.

[ZLS06] J. Zhang, S. Li, and S. Shen. Extracting Minimum Unsatisfiable Cores with a Greedy Genetic Algorithm. In *Proceedings of ACAI*, volume 4304 of *LNCS*, pages 847–856. Springer, 2006.

[ZM02] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of CAV'02*, number 2404 in LNCS, pages 17–36. Springer, 2002.

[ZM03] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of SAT'03*, volume 2919 of *LNCS*. Springer, 2003.

[ZMMM01] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of ICCAD'01*, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.

Index