An Efficient and Trustworthy Theory Solver for Bit-vectors in Satisfiability Modulo Theories

by

Liana Hadarean

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
January 2015

Professor Clark Barrett

Pentru mama, tata și Roxana.

Acknowledgements

I would like to thank my advisor, Clark Barrett, for his patience, guidance and support. This thesis would not have been possible without his help and mentoring. I would also like to thank Cesare Tinelli, for the mentoring he provided especially during my visits to The University of Iowa. I would also like to thank my committee members for their time and feedback: Clark Barrett, Cesare Tinelli, Morgan Deters, Thomas Wies and Benjamin Goldberg.

The work in this thesis is intrinsically linked to the SMT solver CVC4 and it would not have been possible without the hard work and help of the CVC4 development team, especially: Morgan Deters, Dejan Jovanovic, Tim King, Kshitij Bansal and Andrew Reynolds. Working with you truly felt like being part of a team, and the long, sometimes heated, discussions helped me learn so much. I would especially like to thank Morgan Deters for his feedback on the thesis, as well as his endless patience in answering my many questions throughout my years as a PhD student. I would like to also thank Dejan Jovanovic for his insightful suggestions on the bit-vector work, his help in navigating the internals of a SAT solver as well as for being a great office-mate.

Andrew Reynolds introduced me to the LFSC proof language, and helped me understand some (not always intuitive) aspects while helping me with the first draft of the bit-vector proof signature. I would like to thank him for that, and for collaborating with me on my first research project as a PhD student.

Finally, I would like to thank Tim King for his engaged reading of the thesis, insightful comments, but also for being a great friend and helping me get through the intense experience that is thesis writing.

Abstract

As software and hardware systems grow in complexity, automated techniques for ensuring their correctness are becoming increasingly important. Many modern formal verification tools rely on back-end satisfiability modulo theories (SMT) solvers to discharge complex verification goals. These goals are usually formalized in one or more fixed first-order logic theories, such as the theory of fixed-width bit-vectors. The theory of bit-vectors offers a natural way of encoding the precise semantics of typical machine operations on binary data. The predominant approach to deciding the bit-vector theory is via eager reduction to propositional logic. While this often works well in practice, it does not scale well as the bit-width and number of operations increase. The first part of this thesis seeks to fill this gap, by exploring efficient techniques of solving bit-vector constraints that leverage the word-level structure. We propose two complementary approaches: an eager approach that takes full advantage of the solving power of off the shelf propositional logic solvers, and a lazy approach that combines on-the-fly algebraic reasoning with efficient propositional logic solvers. In the second part of the thesis, we propose a proof system for encoding automatically checkable refutation proofs in the theory of bit-vectors. These proofs can be automatically generated by the SMT solver, and act as a certificate for the correctness of the result.

Contents

Dedication iii

Acknowledgements iv

Abstract v

List of Figures x

List of Tables xi

List of Algorithms xii

Introduction 1

1 Preliminaries **4**

1.1 SAT 5

1.2 SMT 7

  1.2.1 Syntax 7

  1.2.2 Semantics 9

1.3 Theory of Bit-vectors 12

  1.3.1 Syntax 13

  1.3.2 Semantics 13

  1.3.3 Sub-theories 20

1.4 CDCL 21

  1.4.1 DPLL 22

1.4.2 CDCL 25
1.4.3 CDCL with assumptions 29
1.5 CDCL(T) 30

2 CVC4 38
2.1 Rewriting 40
2.2 Preprocessing 42
2.3 Justification heuristic 45
2.4 Bit-vector Solvers 47

3 Eager Bit-vector Solving 49
3.1 Architecture 50
3.2 Bit-blasting 53
3.3 AIG Rewriting 55
3.4 Refactoring Isomorphic Circuits 59
3.5 Related Work 62

4 Lazy Bit-vector Solving 66
4.1 Architecture 67
4.2 Bit-blasting Solver 69
4.3 Algebraic Sub-solvers 78
4.3.1 Equality Solver 81
4.3.2 Inequality Solver 82
4.4 In-processing Solver 94
4.5 Lazy Techniques 101
4.6 Model Generation 106

4.7 Related Work 107

5 Lazy vs Eager **109**

5.1 Experimental evaluation 110

5.2 Lazy vs eager 119

5.3 Related work 122

6 Bit-vector Proofs **125**

6.1 LFSC 127

6.1.1 SMT proofs 129

6.1.2 Encoding Resolution 130

6.1.3 Encoding Theory Lemmas 133

6.1.4 CNF Conversion 137

6.2 Proofs in CVC4 140

6.3 Bit-vector Proofs 144

6.3.1 Bit-vector theory LFSC signature 146

6.3.2 Resolution in SAT_{bb} 149

6.3.3 Bit-blasting 150

6.3.4 Rewriting 154

6.4 Related Work 155

7 Conclusion **158**

A LFSC bit-blasting signature code **160**

Bibliography **162**

List of Figures

1.1 Trail for Example 7. 27

1.2 Implication graph for Example 7. 28

2.1 CVC4 architecture. 39

3.1 cvcE architecture. 53

3.2 cvcE vs cvcE+AIG. 56

3.3 cvcE vs cvcE+AIG on brummayerbiere*. 57

3.4 cvcE vs cvcE+AIG on bruttomesso. 57

3.5 cvcE vs cvcE+AIG on spear. 58

3.6 Literals in bit-blasted formula cvcE vs cvcE+RIC on mcm. 63

3.7 Run-time cvcE vs cvcE+RIC on mcm. 63

4.1 cvcLz architecture. 69

4.2 Implication graph for Example 14 (first conflict). 75

4.3 Implication graph for Example 14 (second conflict). 75

4.4 Disabling propagation (cvcLz-P). 79

4.5 Computing eager explanations (cvcLz+EP). 79

4.6 Inequality graph for Example 15. 94

4.7 Disabling equality sub-solver (cvcLz-Eq). 99

4.8 Enabling core solver (cvcLz+core). 99

4.9 Disabling inequality sub-solver (cvcLz-Ineq). 100

4.10 Disabling algebraic in-processing (cvcLz-Alg). 100

4.11 Disabling justification heuristic (cvcLz-J). 104

4.12 Average conflict size using QuickXplain(cvcLz+QX). 105

4.13 QuickXplain conflict minimization (cvcLz+QX). 105

5.1 Comparing cvcLz with cvcE on all of QF_BV. 112

5.2 Comparing cvcLz with cvcE on subset of QF_BV (excluding asp and log-slicing. 112

6.1 Encoding clauses in LFSC. 131

6.2 Encoding propositional resolution in LFSC. 132

6.3 Example resolution proof in LFSC. 133

6.4 Encoding the mapping between $\mathcal{T}$-atoms and their Boolean abstraction in LFSC. 135

6.5 Theory lemma example proof. 138

6.6 Example proof rules for CNF encoding proofs in LFSC. 138

6.7 LFSC CNF conversion proof example. 140

6.8 CVC4 proof module architecture. 144

6.9 Bit-vector resolution proof diagram. 147

6.10 Bit-vector theory LFSC signature. 148

6.11 Bit-blasting proof signature in LFSC. 151

6.12 Encoding $\mathsf{BV}_{\mathsf{bb}}$ conflicts in LFSC. 153

6.13 Rewrite proofs in LFSC. 156

List of Tables

1.1 Propositional logic formulas. 5

1.2 Grammar for Σ-terms. 8

1.3 Bit-vector theory signature. 14

1.4 Bit-vector theory sub-signatures. 21

4.1 LBV sub-solver signatures 80

5.1 Comparing cvcLz and cvcE on QF_BV. 113

5.2 Comparing cvcPll with cvcVBS, cvcLz and cvcE. 114

5.3 Comparing cvcPll with other solvers on QF_BV. 115

5.4 Comparing cvcVBS with other solvers on QF_BV. 116

5.5 Comparing cvcLz with other solvers on QF_AUFBV. 117

List of Algorithms

1 DPLL 24

2 CDCL 26

3 The analyzeConflict procedure. 28

4 CDCL with assumptions. 31

5 CDCL($\mathcal{T}$) 33

6 Bit-blasting bit-wise and. 54

7 Bit-blasting equality. 54

8 Refactoring isomorphic circuits. 62

9 Assumptions conflict. 72

10 CDCL with assumptions and lazy explanations. 77

11 $\mathcal{T}_{\mathsf{BV}}$-check$_{bb}$ 78

12 $\mathcal{T}_{\mathsf{BV}}$-check 80

13 ProcessInequality 84

14 updateModel 85

15 In-processing. 98

16 Conflict minimization minConflict based on quickXplain. 103

Introduction

Designing correct software and hardware systems is an ongoing challenge. Deploying buggy systems can have grave consequences, either monetary or worse. To address this issue formal verification tools employ formal reasoning to give certain guarantees on the correctness of the systems under verification. Such tools include extended static checkers [5,40], bounded and unbounded model-checkers [3,28,49], symbolic execution tools [56], and program verification environments [16, 84] The verification conditions generated by these tools are usually encoded in a formula over a set of fixed first-order logic theories. The task of checking if the formula is satisfiable or not is often delegated to a satisfiability modulo theories (SMT) solver.

One such theory, is the theory of fixed-width bit-vectors. A bit-vector is a sequence of zero or one bits. The bit-vector theory offers a natural way of encoding the semantics of operations that manipulate binary data, the building block almost all modern computer systems. Bit-vectors can model the properties of hardware and software systems, ranging from hardware combinational logic to C/C++ programs.

Many subtle errors arise when a programmer's mental abstraction clashes with the semantics of the implementation, as is often the case with machine arithmetic. For example the assertion $x < x+1$ will always hold for mathematical integers, but may fail for machine integers due to the wrap-around behavior of two's complement arithmetic. Efficiently reasoning about this kind of behavior is essential for detecting subtle bugs.

A significant part of this thesis explores approaches to efficiently solving bit-vector constraints. Decision procedures for the bit-vector theory have traditionally relied on eager reduction to propositional logic via a technique vividly known as bit-blasting. The first part of this thesis focuses on the features of the eager solver cvcE implemented in the SMT solver CVC4. In particular we discuss a new technique of reducing the size of the bit-blasted formula by factoring out isomorphic sub-circuits.

While often efficient in practice, the eager bit-blasting approach does not scale well when algebraic operators over large bit-width bit-vectors are involved. One reasons is that bit-blasting loses the word-level structure and word-level simplifications can only be applied before solving. The second part of this thesis, explores an alternative to eager bit-blasting, a lazy bit-blasting approach to solving bit-vector constraints based on the DPLL($\mathcal{T}$) SMT solver framework. We propose several novel techniques within this lazy framework: (i) a dedicated SAT solver for bit-vector theory that supports bit-blasting-based propagation with lazy explanations; (ii) specialized bit-vector sub-solvers that reason about fragments of the bit-vector theory, and (iii) inprocessing techniques to reduce the size of the bit-blasted formula when possible. These features significantly improve performance and enable us to solve hard problems no other solvers can.

SMT solvers are at the heart of many formal verification tools. Guaranteeing the correctness of the SMT solvers' results is a longstanding concern. The third part of the thesis, describes an approach to increasing the trustworthiness of SMT solvers by instrumenting the solver to emit an externally-checkable certificate of correctness in the form of a proof of unsatisfiability. The combination of algebraic and propositional reasoning in the bit-vector theory makes bit-vector proofs particularly challenging. To address this. we propose a machine checkable proof system for proofs in the bit-vector theory.

The thesis is structured as follows. Chapter 1 introduces background information on SAT and SMT that will be referenced throughout the thesis. Chapter 2 describes the architecture of the SMT solver CVC4, while highlighting the features essential to bit-vector solving. The features of the eager bit-vector solver are covered in Chapter 3, and those of the lazy bit-vector solver in Chapter 4. Chapter 5 evaluates the performance of the two approaches compared to each other, as well as compared to other state-of-the-art SMT solvers. This chapter also illustrates the complementary nature of the two approaches. The proof infrastructure of CVC4 as well as the proof system for encoding SMT generated bit-vector proofs are described in Chapter 6. Finally, we conclude by summarizing the contributions of the thesis in Chapter 7.

Chapter 1

Preliminaries

This chapter gives a brief introduction to concepts that will be referenced later in the thesis. While attempting to be self-contained, it assumes familiarity with the topics and is intended to serve as a refresher.

Section 1.1 first gives a brief introduction to propositional logic which is then generalized to first-order logic in Section 1.2. Here we also define theories and introduce the notion of satisfiability modulo theories (SMT). Using these formalisms, we introduce the theory of fixed-width bit-vectors in Section 1.3.

The following section, Section 1.4, describes a state-of-the-art procedure for deciding propositional logic and highlights the aspects that will be relevant later in the thesis. Finally, Section 1.5 describes a framework that extends the propositional logic decision procedure to a decision procedure for first-order logic modulo theories.

$$
\begin{array}{rcll}
a & ::= & v & v \in V \\
& | & \bot \mid \top & \\
& | & \neg a \mid a \wedge a &
\end{array}
$$

Table 1.1: Propositional logic formulas.

1.1 SAT

Propositional logic is concerned with reasoning about the truth value of propositions: "statements" that can be either *true* or *false*. More complex propositions can be built by using Boolean connectives such as and and not. We will make this definition precise by first giving the syntax of propositional logic and then its semantics.

Fix an infinite set of variable symbols V to represent the *propositional variables*. Also fix the symbols $\top$ and $\bot$ to represent the truth values *true* and *false* respectively. *Propositional formulas* are built from propositional variables and the $\top$ and $\bot$ constants using *Boolean connective* symbols $\wedge$ (and) and $\neg$ (not). Table 1.1 gives the syntax of propositional logic by showing the grammar for constructing propositional formulas.

A *propositional assignment* μ is a map from Boolean variables V to Boolean values, $\mu : V \to \{true, false\}$. Given a propositional assignment μ we can generalize it to evaluate arbitrary Boolean formulas by defining its homomorphic extension $[\![\_]\!]_\mu$. We define $[\![\_]\!]_\mu$ inductively as follows:

- $[\![x]\!]_\mu = \mu(x)$ for $x \in V$
- $[\![\top]\!]_\mu = true$
- $[\![\bot]\!]_\mu = false$
- $[\![\neg a]\!]_\mu = true$ iff $[\![a]\!]_\mu = false$
- $[\![a_1 \wedge a_2]\!]_\mu = true$ iff $[\![a_1]\!]_\mu = true$ and $[\![a_2]\!]_\mu = true$

For convenience, we introduce other Boolean connectives, all of which could be expressed in terms of the two already introduced: $\vee$ (or), $\Rightarrow$ (implies), $\Leftrightarrow$ (if-and-only-if), $\oplus$ (exclusive or) and ite (if-then-else). The semantics of these other operators is given below in terms of $\wedge$ and $\neg$:

- $a_1 \vee a_2 = \neg(\neg a_1 \wedge \neg a_2)$: at least one of a_1 or a_2 is *true*.
- $a_1 \Rightarrow a_2 = \neg(a_1 \wedge \neg a_2)$: if a_1 is *true* so must a_2.
- $a_1 \Leftrightarrow a_2 = (a_1 \Rightarrow a_2) \wedge (a_2 \Rightarrow a_1)$: a_1 and a_2 have the same truth value.
- $a_1 \oplus a_2 = \neg(a_1 \Leftrightarrow a_2)$: a_1 and a_2 do not have the same truth value.
- $ite(c, a_1, a_2) = (c \Rightarrow a_1) \wedge (\neg c \Rightarrow a_2)$: same truth value as a_1 if c holds and a_2 otherwise.

A Boolean assignment μ *satisfies* a propositional formula a if a evaluates to *true* under the assignment: $[\![a]\!]_\mu = true$. If there is no such assignment, we say a is *unsatisfiable*. A formula is *valid* (or a *tautology*) if evaluates to *true* under all possible assignments. The problem of propositional satisfiability (SAT) is that of deciding whether a given propositional formula is satisfiable or not. SAT is a well studied problem and one of the classical NP-complete problems [29].

Definition 1. A *literal* is a Boolean variable or its negation. We say a literal has *negative polarity* if it is a negated variable and *positive polarity* otherwise. A *clause* is a disjunction (or) of literals. We say a formula is in *conjunctive normal form* (CNF) if it is a conjunction (and) of clauses.

Most decision procedures for SAT assume the input formula is in CNF. There are known algorithms that can convert an arbitrary Boolean formula to an equisatisfiable formula in CNF in linear time [81] by introducing intermediate variables.

Example 1. The following formula is in CNF:

$$\begin{array}{ll} & (v_1 \vee \neg v_2 \vee \neg v_3) \\ \wedge & (v_2 \vee v_1) \\ \wedge & \neg v_1 \end{array}$$

where v_1, v_2, and v_3 are Boolean variables, $v_1, \neg v_1, v_2, \neg v_2, v_3, \neg v_3$ are the literals and $v_1 \vee \neg v_2 \vee \neg v_3$, $v_2 \vee v_1$, and $\neg v_1$ are clauses.

We will sometimes represent a CNF formula as a set of clauses and a clause as a set of literals.

1.2 SMT

First-order logic (FoL) extends propositional logic with function symbols and variables that can now range over more expressive domains. In this section, we will focus on multi-sorted first-order logic with equality. For a more detailed and formal introduction see [38,63]. [1]

1.2.1 Syntax

Fix an infinite set of sort symbols S and a disjoint infinite set of variable symbols X each uniquely associated with a sort $\sigma \in S$. We write $x : \sigma$ to express that variable $x \in X$ has sort σ.

Definition 2. A *signature* Σ consists of the following:

---

[1] The presentation of FOL in this chapter differs from that of [38] by having the Boolean sort explicit. While most presentations distinguish between terms and formulas, we will define formulas as terms of Boolean sort. This allows for function symbols to take Boolean variables as arguments, a common feature in SMT.

$$\begin{array}{lcll}
t & ::= & c & c \in \Sigma^F \text{ and } c^\# = 0 \\
 & \mid & x & x \in X \\
 & \mid & f(t_1, \ldots, t_n) & f \in \Sigma^F,\, f^\# = n,\, \tau(f) = (\sigma_1, \ldots, \sigma_n, \sigma) \text{ and } \tau(t_i) = \sigma_i \\
 & \mid & ite_\sigma(\phi, t_1, t_2) & f \in \Sigma^F,\, \tau(\phi) = Bool \text{ and } \tau(t_1) = \tau(t_2) = \sigma \\
 & \mid & t_1 =_\sigma t_2 & \tau(t_1) = \tau(t_2) = \sigma \\
 & \mid & \forall x : \sigma . \varphi & x \in X,\, x : \sigma \text{ and } \tau(\varphi) = Bool \\
 & \mid & \top \mid \bot & \\
 & \mid & \neg \varphi & \tau(\varphi) = Bool \\
 & \mid & \varphi_1 \wedge \varphi_2 & \tau(\varphi_1) = Bool \text{ and } \tau(\varphi_2) = Bool
\end{array}$$

Table 1.2: Grammar for Σ-terms.

- a set of *sort symbols* $\Sigma^S \subseteq S$
- a set of *function symbols* Σ^F
- an *arity* mapping $(\_)^\#$ that maps each function symbol to an arity $n \geq 0$.
- a *sort* mapping τ that uniquely maps each function symbol $f \in \Sigma^F$ to $\tau(f) = (\sigma_1, \ldots, \sigma_n, \sigma)$ for $f^\# = n$.

Each signature will also include by default the Boolean sort symbol $Bool$, and symbols $=_\sigma$ (equality) and ite_σ (if-then-else) for each sort $\sigma \in sorts$. When the arity of a function is 0 we can think of it as a function that takes no arguments. We will call such 0-arity function symbols *constants*.

The Boolean constants $\top, \bot$, the Boolean connectives $\neg, \wedge$, and the ite and equality symbols are mapped to the following fixed sorts by τ in all signatures Σ:

- $\tau(\top) = \tau(\bot) = Bool$
- $\tau(\wedge) = (Bool, Bool, Bool)$
- $\tau(\neg) = (Bool, Bool)$
- $\tau(=_\sigma) = (\sigma, \sigma, Bool)$

- $\tau(ite_\sigma) = (Bool, \sigma, \sigma, \sigma)$

We extend the sort mapping τ to applications of function symbols as follows:

$$\tau(f(t_1, \ldots, t_n)) = \sigma \text{ where } f \in \Sigma^F,\ f^{\#} = n,\ \tau(f) = (\sigma_1, \ldots, \sigma_n, \sigma) \text{ and } \tau(t_i) = \sigma_i$$

Intuitively, τ maps a well-sorted function application to its return sort.

The grammar in Figure 1.2 shows how to build well-formed Σ-terms. For the equality $=_\sigma$ and ite_σ symbols we will omit the sort σ when it is obvious from context. We will refer to Σ-terms of sort $Bool$ as Σ-formulas. A Σ-atom is a special kind of Σ-formula that is either a variable, a constant, a quantified formula, the application of a function symbol other than the Boolean connectives or an equality between non-Boolean terms. Intuitively atoms are formulas with no Boolean structure. Σ-literals are Σ-atoms or negations of Σ-atoms and Σ-clauses are disjunctions of Σ-literals.

Example 2. The signature for Presburger arithmetic is: $\Sigma_{\mathbb{Z}} = (Int, 0, 1, +, -, <)$ where Int is the integer sort, 0 and 1 are constant symbols, $+, -, <$ are function symbols (though part of the signature, we do not list the built-in symbols). In this signature, $x < 1$ is an atom and $x < 1 \vee x = y + 1$ is a formula, for variables x and y.

1.2.2 Semantics

We give meaning to first-order logic formulas using the notion of *models*. A model maps each sort symbol to a set of values and the function symbols to operations over these values.

Definition 3. Given a signature Σ, a Σ-model $\mathcal{M}$ is defined as follows:

- For each $\sigma \in \Sigma^S$, $\mathcal{M}$ associates a non-empty set A_σ called the domain of σ in $\mathcal{M}$. The $Bool$ sort always maps to the set $A_{Bool} = \{false, true\}$ with $false \neq true$.
- For each function symbol $f \in \Sigma^F$ with $\tau(f) = (\sigma_1, \ldots, \sigma_n, \sigma)$ and $f^\# = n$, $\mathcal{M}$ associates $f^\mathcal{M}$ where $f^\mathcal{M} : (A_{\sigma_1} \times \ldots \times A_{\sigma_n}) \rightarrow A_\sigma$.

An *assignment* μ on a fixed Σ-model is a map from the set of variables X to values in their respective domains: $\mu(x) = v$ for $v \in A_\sigma$ and $x : \sigma$.

Given a Σ-model $\mathcal{M}$ and an assignment μ, we define the evaluation mapping $[\![\_]\!]_\mu^\mathcal{M}$ as follows:

- For variable $x \in X$, $[\![x]\!]_\mu^\mathcal{M} = \mu(x)$.
- For applications of function f, we have:

$$[\![f(t_1, \ldots, t_2)]\!]_\mu^\mathcal{M} = f^\mathcal{M}([\![t_1]\!]_\mu^\mathcal{M}, \ldots, [\![t_2]\!]_\mu^\mathcal{M}).$$

- For quantified formulas $[\![\forall x : \sigma.\varphi]\!]_\mu^\mathcal{M} = true$ iff for all $v \in A_\sigma$, $[\![\varphi]\!]_{\mu'}^\mathcal{M} = true$, where μ' is equivalent to μ except that it maps x to v.
- For equalities, we have $[\![t_1 =_\sigma t_2]\!]_\mu^\mathcal{M} = true$ if $[\![t_1]\!]_\mu^\mathcal{M} = [\![t_n]\!]_\mu^\mathcal{M}$ and otherwise $[\![t_1 =_\sigma t_2]\!]_\mu^\mathcal{M} = false$.
- For if-then-else terms $[\![ite(\varphi, t_1, t_2)]\!]_\mu^\mathcal{M} = [\![t_1]\!]_\mu^\mathcal{M}$ if $[\![\varphi]\!]_\mu^\mathcal{M} = true$ and otherwise $[\![ite(\varphi, t_1, t_2)]\!]_\mu^\mathcal{M} = [\![t_2]\!]_\mu^\mathcal{M}$.
- For the propositional constants, we have $[\![\bot]\!]_\mu^\mathcal{M} = false$ and $[\![\top]\!]_\mu^\mathcal{M} = true$.
- For propositional negation $[\![\neg t]\!]_\mu^\mathcal{M} = true$ if $[\![t]\!]_\mu^\mathcal{M} = false$ and $[\![\neg t]\!]_\mu^\mathcal{M} = false$ otherwise.

- For the propositional conjunction $[\![t_1 \wedge t_2]\!]^{\mathcal{M}}_{\mu} = true$ iff $[\![t_1]\!]^{\mathcal{M}}_{\mu} = true$ and $[\![t_2]\!]^{\mathcal{M}}_{\mu} = true$.

Although we introduced propositional logic in a slightly different way, first-order logic subsumes propositional logic. By restricting the signature to only the Boolean symbols: ($\wedge$, $\neg$, $\top$, $\bot$) and variables of sort $Bool$, we can embed propositional logic in FoL.

Definition 4. A Σ-interpretation $\mathcal{I}$ is a pair $(\mathcal{M}, \mu)$ where $\mathcal{M}$ is a Σ-model and μ is a variable assignment on $\mathcal{M}$.

We can formally define the notion of *satisfiability* in first-order logic using interpretations:

- An interpretation $\mathcal{I} = (\mathcal{M}, \mu)$ *satisfies* a Σ-formula φ ($\mathcal{I} \models \varphi$) if $[\![\varphi]\!]^{\mathcal{M}}_{\mu} = true$.
- A Σ-formula ϕ *entails* Σ-formula φ ($\phi \models \varphi$) if for all interpretations $\mathcal{I}$, if $\mathcal{I} \models \phi$ then also $\mathcal{I} \models \varphi$.
- A Σ-formula φ is *unsatisfiable* ($\varphi \models \bot$) if there is no $\mathcal{I}$ that satisfies it.
- A Σ-formula φ is *valid* ($\models \varphi$) if for all Σ-interpretations $\mathcal{I}$ the following holds $\mathcal{I} \models \varphi$.

We are often not interested in all possible models of a signature. For example we may want to interpret the function symbol $+$ as mathematical addition. Such specific models can be formalized using the notion of *theories*. Fixing the model not only allows us to reason about domains of interest, but also enables the use of more efficient decision procedures specialized for these domains.

We formally define a *theory* $\mathcal{T}$ as a class of models with the same signature.[2]

---

[2] We are following the definition in [10] which is more general than the standard first-order logic definition.

Definition 5. A Σ-theory $\mathcal{T}$ is a pair (Σ, A) where Σ is a signature and A is a class of Σ-models.

A $\mathcal{T}$-interpretation $\mathcal{I} = (\mathcal{M}, \mu)$ for a theory $\mathcal{T} = (\Sigma, \mathsf{A})$ is a Σ-interpretation such that $\mathcal{M} \in \mathsf{A}$.

The notions of satisfiability from first-order logic can naturally be extended to *satisfiability modulo theories* (SMT) over $\mathcal{T}$-formulas:

- We say that a $\mathcal{T}$-formula φ is *$\mathcal{T}$-satisfiable* in theory $\mathcal{T}$ if there exists a $\mathcal{T}$-interpretation $\mathcal{I}$ such that $\mathcal{I} \models \varphi$.
- A $\mathcal{T}$-formula φ is *$\mathcal{T}$-unsatisfiable* written $\varphi \models_{\mathcal{T}} \bot$, if there is no $\mathcal{T}$-interpretation $\mathcal{I}$ that satisfies φ.
- A $\mathcal{T}$-formula φ is *$\mathcal{T}$-valid* written $\models_{\mathcal{T}} \varphi$, if for all $\mathcal{T}$-interpretations $\mathcal{I}$ we have $\mathcal{I} \models \varphi$
- A $\mathcal{T}$-formula ϕ *$\mathcal{T}$-entails* $\mathcal{T}$-formula φ written $\phi \models_{\mathcal{T}} \varphi$ if for all $\mathcal{T}$-interpretations $\mathcal{I}$ if $\mathcal{I} \models \phi$ we have $\mathcal{I} \models \varphi$.

The notions of atoms, literals and clauses naturally extend to $\mathcal{T}$-atoms, $\mathcal{T}$-literals and $\mathcal{T}$-clauses. A *$\mathcal{T}$-constraint* is a conjunction of $\mathcal{T}$-literals.

We are concerned with the *constraint satisfiability* problem for a theory $\mathcal{T}$, which consists of deciding whether a $\mathcal{T}$-constraint is *$\mathcal{T}$-satisfiable*, that is if there exists a $\mathcal{T}$-interpretation $\mathcal{I}$ that satisfies the constraint.

1.3 Theory of Bit-vectors

Almost all computing devices operate by manipulating finite sequences of zeros and ones - *bit-vectors*. The *bit-vector theory* offers a natural way of encoding these operations. It supports low-level operations such as extracting individual bits, as well as arithmetic operations where the bit-vector is interpreted as a number.

In this section, we introduce the *theory of fixed-width bit-vectors*: $\mathcal{T}_{\mathsf{bv}}$. First we present it in terms of the formalisms introduced in the previous section and then show examples to give the intuition (or lack-of) behind its semantics.

1.3.1 Syntax

Table1.3 gives the signature Σ_{bv} for the bit-vector theory. The set of sorts Σ^S_{bv} contains infinitely many sort symbols $[n]$ where n is a strictly positive natural number. Note that, except for the constants, the function symbols in Table 1.3 are overloaded; for example, $+$ stands for any of the symbols in the infinite family $\{+ : ([n], [n], [n])\}_{n>0}$. For simplicity, we restrict our attention to a subset of the bit-vector operators described in the SMT-LIB v2.0 standard [9]; the missing ones can easily be expressed in terms of those given here. To illustrate that a bit-vector term t is of sort $[n]$ we will write $t_{[n]}$ and omit the subscript when it is obvious from context.

1.3.2 Semantics

We will give the semantics of the theory in terms of the standard model for the theory of bit-vectors $\mathcal{BV}$.

A natural way to think of a bit-vectors is as a fixed-length sequence of binary bits. The domain of our model values will be the 0 and 1 bit values.[3] For example 010101 is a bit-vector of length 6. An alternative, but equivalent view is that of a function between an index and the bit value at that index. The previous example can be represented as a function that returns 0 for all odd indices and 1 for the even ones. While the sequence

---

[3] Not to be confused with the corresponding integer numbers or the $\mathcal{T}_{\mathsf{bv}}$ signature symbols 0 and 1.

	Symbol	Name
Σ_{bv}^{S}	$[1]$	bit-vector sort of length 1
	$[2]$	bit-vector sort of length 2
	$\vdots$	$\ldots$

	Symbol	Name	$(\_)^{\#}$	Sort map τ
Σ_{bv}^{F}	$0_{[1]}, 1_{[1]}$	constants	0	$[1]$
	$00_{[2]}, 01_{[2]} \ldots$		0	$[2]$
	$\vdots$		0	$\vdots$
	$\_ \circ \_$	concat	2	$([n], [m], [n+m])$
	$\_[i:j]$	extract	1	$([n], [j-i+1])$ for $n > i > j \geq 0$
	$\sim \_$	bitwise not	1	$([n], [n])$
	$\_\&\_$	bitwise and	2	$([n], [n], [n])$
	$\_ \mid \_$	bitwise or	2	$([n], [n], [n])$
	$\_ + \_$	plus	2	$([n], [n], [n])$
	$-\_$	bvneg	1	$([n], [n])$
	$\_ \times \_$	times	2	$([n], [n], [n])$
	$\_ \div \_$	div	2	$([n], [n], [n])$
	$\_\%\_$	remainder	2	$([n], [n], [n])$
	$\_ >> \_$	right shift	2	$([n], [n], [n])$
	$\_ << \_$	left shift	2	$([n], [n], [n])$

	Symbol	Name	$(\_)^{\#}$	Sort map τ
Σ_{bv}^{P}	$\_ =_{[n]} \_$	equal	2	$([n], [n], Bool)$
	$\_ < \_$	less-than	2	$([n], [n], Bool)$
	$\_ \leq \_$	less-than-or-equal	2	$([n], [n], Bool)$

Table 1.3: Bit-vector theory signature.

view is more intuitive, the function one makes it easier to give a precise definition of the various operators. Therefore we will adopt the latter for defining the semantics of the theory. We will switch between the two views depending on which is the clearest and most convenient in the context.

We represent a bit-vector of length n as a lambda term of the form:

$$\lambda x : [0, n).f(x)$$

where $f : [0, n) \to \{0, 1\}$ and $[0, n)$ denotes the set of natural numbers $\{0, 1, \ldots n - 1\}$. Note that a bit-vector of size n is represented by a *total* function with the domain consisting only of the valid indices.

Example 3. The function representation of 0101010101 is:

$$\begin{aligned} \lambda x : [0, 8). \quad & \text{if } (x \text{ odd}) \text{ then } 1 \\ & \qquad\qquad\quad \text{else } 0 \end{aligned}$$

Another way to interpret a bit-vector is as a 2's complement number. For example 0101010101 corresponds to $1365_{[8]}$. Let $\mathsf{bitToNat}$ be a map from bits to natural numbers defined as follows:

$$\mathsf{bitToNat}(b) = \text{if } b = 1 \text{ then } 1 \text{ else } 0.$$

Using $\mathsf{bitToNat}$ we define the following functions to easily convert between bit-vectors and their corresponding natural number, $\mathsf{natBv}_n : \mathbb{N} \to [n]$ and $\mathsf{bvNat} : [n] \to [0, 2^n)$:

$$\begin{aligned} \mathsf{natBv}_n(x) &= \lambda i : [0, n).\text{if } (x \div 2^i) \bmod 2 = 0 \text{ then } 0 \text{ else } 1 \\ \mathsf{bvNat}(f) &= \sum_{i=0}^{n-1} \mathsf{bitToNat}(f(i)) \cdot 2^i \end{aligned}$$

We use $+$, $\times$, $\div$ and $\%$ to refer to both the symbols in the bit-vector signature Σ_{bv} and the integer number arithmetic operations of addition, multiplication, truncating division and remainder. Which one we are referring to should be obvious from the context.

For brevity we will write $[\![\_]\!]$ to refer to $[\![\_]\!]^{\mathcal{BV}}_{\mu}$ for the remainder of the chapter, for some variable assignment μ. Using the constructs above, we can formally define the bit-vector model $\mathcal{BV}$ in terms of its evaluation mapping $[\![\_]\!]$ as follows:

- For each $[n] \in \Sigma^{S}_{\mathsf{bv}}$, we have $A_{[n]} = \{\lambda x : [0, n).f(x) \mid f : [0, n) \to \{\mathtt{0}, \mathtt{1}\}\}$.
- For the constant symbols, we have $[\![0_{[1]}]\!] = \lambda x : [0, 1).\mathtt{0}$, $[\![1_{[1]}]\!] = \lambda x : [0, 1).\mathtt{1}$ and so forth.
- For function symbols:

$$
\begin{aligned}
[\![a_{[n]} \circ b_{[m]}]\!] &= \lambda x : [0, n+m).\text{if } x < m \text{ then } [\![b]\!](x) \text{ else } [\![a]\!](x-m) \\
[\![a_{[n]}[i:j]]\!] &= \lambda x : [0, i-j+1).[\![a]\!](j+x) \\
[\![\sim a_{[n]}]\!] &= \lambda x : [0, n).\text{if } [\![a]\!](x) = \mathtt{0} \text{ then } \mathtt{1} \text{ else } \mathtt{0} \\
[\![a_{[n]} \& b_{[n]}]\!] &= \lambda x : [0, n).\text{if } [\![a]\!](x) = \mathtt{1} \text{ and } [\![b]\!](x) = \mathtt{1} \text{ then } \mathtt{1} \text{ else } \mathtt{0} \\
[\![a_{[n]} \mid b_{[n]}]\!] &= \lambda x : [0, n).\text{if } [\![a]\!](x) = \mathtt{1} \text{ or } [\![b]\!](x) = \mathtt{1} \text{ then } \mathtt{1} \text{ else } \mathtt{0} \\
[\![a_{[n]} + b_{[n]}]\!] &= \mathsf{natBv}_n(\mathsf{bvNat}([\![a]\!]) + \mathsf{bvNat}([\![b]\!])) \\
[\![-a_{[n]}]\!] &= \mathsf{natBv}_n(2^n - \mathsf{bvNat}([\![a]\!])) \\
[\![a_{[n]} \times b_{[n]}]\!] &= \mathsf{natBv}_n(\mathsf{bvNat}([\![a]\!]) \times \mathsf{bvNat}([\![b]\!])) \\
[\![a_{[n]} \div b_{[n]}]\!] &= \text{if } \mathsf{bvNat}([\![b]\!]) \neq 0 \text{ then } \mathsf{natBv}_n(\mathsf{bvNat}([\![a]\!]) \div \mathsf{bvNat}([\![b]\!])) \\
[\![a_{[n]} \% b_{[n]}]\!] &= \text{if } \mathsf{bvNat}([\![b]\!]) \neq 0 \text{ then } \mathsf{natBv}_n(\mathsf{bvNat}([\![a]\!]) \% \mathsf{bvNat}([\![b]\!])) \\
[\![a_{[n]} >> b_{[n]}]\!] &= \mathsf{natBv}_n(\mathsf{bvNat}([\![a]\!]) \div 2^{\mathsf{bvNat}([\![b]\!])}) \\
[\![a_{[n]} << b_{[n]}]\!] &= \mathsf{natBv}_n(\mathsf{bvNat}([\![a]\!]) \times 2^{\mathsf{bvNat}([\![b]\!])}) \\
[\![a_{[n]} =_{[n]} b_{[n]}]\!] &= \text{if } \mathsf{bvNat}([\![a]\!]) = \mathsf{bvNat}([\![b]\!]) \text{ then } \mathit{true} \text{ else } \mathit{false}
\end{aligned}
$$

$$
\begin{aligned}
[\![a_{[n]} < b_{[n]}]\!] &= \text{if } \mathsf{bvNat}([\![a]\!]) < \mathsf{bvNat}([\![b]\!]) \text{ then } true \text{ else } false \\
[\![a_{[n]} \leq b_{[n]}]\!] &= \text{if } \mathsf{bvNat}([\![a]\!]) \leq \mathsf{bvNat}([\![b]\!]) \text{ then } true \text{ else } false
\end{aligned}
$$

Note that the semantics of the division operations $\div$ and $\%$ are only defined when the divisor is not $0_{[n]}$. This is in accordance with the SMT-LIB v2.0 standard. The concatenation $\_ \circ \_$ and extract $\_[i : j]$ operations are the only ones that can alter the length of bit-vectors. The Boolean operators $\sim$, $\&$ and $|$ are the bit-wise equivalent of the propositional logic operators $\neg$, $\wedge$ and $\vee$ if we map 0 to *false* and 1 to *true*. Other bit-wise Boolean operators such as $\oplus$ can be expressed using the given ones.

The arithmetic operations interpret the bit-vector as a base-2 natural number and are equivalent to the corresponding $\bmod 2^n$ arithmetic operations, where n is the size of the bit-vectors. Note that this does not always match the semantics of natural number arithmetic, as highlighted by the following example.

Example 4. Consider the following formula $x < y \Rightarrow x < y + 1$ in the theory of integers. It is not hard to see that the formula is valid. However, the bit-vector equivalent $x_{[n]} < y_{[n]} \Rightarrow x_{[n]} < y_{[n]} + 1_{[n]}$ is not. Let us fix $n = 8$. Consider the model that assigns $x_{[8]} = 1_{[8]}$ and $y_{[8]} = 255_{[8]}$. The bit representation of $y_{[8]}$ is 11111111 (or equivalently $\lambda i : [0, 7).1$). Computing the sum $y_{[8]} + 1_{[8]}$:

$$
\begin{array}{rcl}
11111111 & + & 255_{[8]} \\
\underline{\;\;00000001} & & 1_{[8]} \\
100000000 & & 0_{[8]}
\end{array}
$$

However since the resulting bit-vector only has 8 bits an *overflow* occurs and the top bit is dropped and $y_{[n]} + 1_{[n]} = 0_{[n]}$, which is not greater than $1_{[n]}$. The formula $x_{[n]} < y_{[n]} \Rightarrow x_{[n]} + 1_{[n]} \leq y_{[n]}$ is valid in the bit-vector theory and so is its integer equivalent.[4]

So far we have only defined unsigned arithmetic operations that interpret the bit-vectors as positive integers. We can express signed arithmetic on bit-vectors by using 2's complement arithmetic using the following two functions:

$$\begin{aligned}
\mathsf{intBv}_n(x) &= \quad \text{if } x \geq 0 \quad 0_{[1]} \circ \mathsf{natBv}_{n-1}(x) \\
&\quad\ \text{elif } x \geq -2^{n-1} \quad 1_{[1]} \circ \mathsf{natBv}_{n-1}(2^n + x) \\
&\quad\ \text{else} \quad \mathsf{natBv}_n(x \%_E 2^n) \\
\mathsf{bvInt}(f) &= -2^{n-1} \cdot \mathsf{bitToNat} f(n-1) + \sum_{i=0}^{n-2} f(i) \cdot 2^i
\end{aligned}$$

where $\%_E$ represents Euclidian division (the remainder is always positive). Addition and multiplication do not have a signed counterpart as they operate the same on both interpretations.

The signed version of $<$, _ $\div$ _ and _$\%$_, $<_S$, _ $\div_S$ _ and _$\%_S$_ do have different semantics:

$$\begin{aligned}
[\![a_{[n]} <_S b_{[n]}]\!] &= \text{if } \mathsf{bvInt}([\![a]\!]) < \mathsf{bvInt}([\![b]\!]) \text{ then } true \text{ else } false \\
[\![a_{[n]} \div_S b_{[n]}]\!] &= \text{if } \mathsf{bvInt}([\![b]\!]) \neq 0 \text{ then } \mathsf{intBv}_n(\mathsf{bvInt}([\![a]\!]) \div \mathsf{bvInt}([\![b]\!])) \\
[\![a_{[n]} \%_S b_{[n]}]\!] &= \text{if } \mathsf{bvInt}([\![b]\!]) \neq 0 \text{ then } \mathsf{intBv}_n(\mathsf{bvInt}([\![a]\!]) \% \mathsf{bvInt}([\![b]\!]))
\end{aligned}$$

The $\div$ and $\%$ operators on integers represent truncated division. Bit-vector signed division $\div_s$ always returns the same value as integer division, except when dividing $-2^n_{[n]}$ by $-1_{[n]}$ which leads to an overflow. Note that we can express these signed operations in terms of the unsigned ones, by just examining the sign bits of the operands. For example $<_S$ can be expressed in terms of the unsigned $<$ operator as follows:

---

[4] Example adapted from the SMT-LIB 2014-06-03 QF_BV pspace family of benchmarks.

$$\begin{aligned} a_{[n]} <_S b_{[n]} \quad = \quad & \text{if} \quad a[n-1] = b[n-1] \text{ then } a[n-2:0] < b[n-2:0] \\ & \text{else} \quad b[n-1] < a[n-1] \end{aligned}$$

The division operators can be translated in a similar way, albeit more involved. See the SMT-LIB v2.0 `QF_BV` logic description for the details. For simplicity, we will mostly focus on the unsigned operators from now on.

Example 5. Consider the following bit-vector formula:

$$0_{[n]} \leq_S (ite(x_{[n]} <_S 0_{[n]}, -x_{[n]}, x_{[n]}))$$

Its integer counterpart is clearly valid. Consider the case when $x_{[n]} = -2^{n-1}_{[n]}$, the smallest negative number that can be represented in n bits. We have:

$$\begin{aligned} -x_{[n]} &= \sim (10\ldots0) + 00\ldots1 \\ &= 01\ldots1 + 00\ldots1 \\ &= 10\ldots0 \\ &= -2^{n-1}_{[n]} \end{aligned}$$

which is a negative number. This counter-intuitive result is due to the fact that the absolute value of the largest negative number that can be stored in n bits (-2^{n-1}) is bigger than the largest positive number that can be stored in n bits ($2^{n-1} - 1$).[5]

As shown above, the differences between the semantics of mathematical integers and machine integers as encoded by bit-vectors are often counter-intuitive. Subtle bugs arise when programmers intuitively think in terms of mathematical integers.

---

[5] This is why the `abs` C function is undefined for the most negative integer.

It is not hard to see that the decision procedure for the bit-vector theory is at least as hard as propositional logic (NP-complete): we can encode propositional logic in bit-vectors by using bit-vectors of size 1 and the Boolean operators.

The reverse direction is a bit more subtle. Bit-vector satisfiability can be reduced to SAT by replacing function symbols by their hardware circuit representation expressed in propositional logic. The process of encoding a bit-vector formula into propositional logic is called *bit-blasting*. Bit-vector addition for example can be modeled using a ripple-carry adder circuit. We will make this concrete in Section 3.2. However, the reduction to SAT is not always polynomial when considering a binary encoding of the bit-width. The decision problem for the quantifier-free fixed-width bit-vector theory has been shown to be NEXPTIME-complete [57].

1.3.3 Sub-theories

One of the main difficulties in reasoning algebraically about the bit-vector theory is the fact that it combines bit-fiddling operations with arithmetic operations. The semantic relationships between these operations are complex and not easily axiomatizable.

Example 6. The following bit-vector constraint combines bit-wise Boolean operations and arithmetic to encode that $x_{[n]}$ is a power of 2 [83]:

$$(x_{[n]} - 1_{[n]}) \& x_{[n]} = 0_{[n]}$$

This is not immediately obvious or intuitive.

To reign in this complexity, we can think of $\mathcal{T}_{\mathsf{bv}}$ as consisting of several (non-disjoint) sub-theories. We partition the bit-vector signature into the sub-signatures in Table 1.4.

Sub-signature	Symbols	Name
Σ_{eq}	$=_{[n]}, 0_{[1]}, 1_{[1]} \ldots$	equality
Σ_{con}	$\circ, \_[i : j]$	core
Σ_{ineq}	$<, \leq$	inequality
Σ_{ari}	$+, -, \times, \div, \%$	arithmetic
Σ_{bool}	$\sim, \vert, \&,$	Boolean
Σ_{shift}	$>>, <<$	shift

Table 1.4: Bit-vector theory sub-signatures.

Using these sub-signatures we can define the sub-theories corresponding to the following signatures: Σ_{eq}, $\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{con}}$, $\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{ineq}}$, $\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{ari}}$, $\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{bool}}$, and $\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{shift}}$.

The $\mathcal{T}_{\mathsf{bv}}$-satisfiability of conjunctions of equalities between terms over the *core* sub-signature $\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{con}}$ is decidable in polynomial time [24, 30]. However, adding almost any of the additional operators, or allowing for arbitrary Boolean structure, makes the $\mathcal{T}_{\mathsf{bv}}$-satisfiability problem NP-hard [11].

1.4 CDCL

Although the satisfiability problem in propositional logic (SAT) is NP-complete, modern SAT solvers can routinely decide problems with millions of clauses. They do so by exploiting the problem structure to efficiently prune the search space. In this section, we will describe one of the state-of-the-art SAT algorithms: conflict-driven clause learning (CDCL). We first present the Davis-Putnam-Logemann-Loveland (DPLL) SAT algorithm in 1.4.1, the precursor of CDCL. In 1.4.2 we show how CDCL extends DPLL, and in 1.4.3 we present an extension of CDCL to allow incremental reasoning. All algorithms in this section operate on formulas in CNF by converting the input propositional formula ψ to an *equisatisfiable* formula C using the toCnf procedure [81].

1.4.1 DPLL

A naive brute-force algorithm for SAT could just enumerate all possible truth assignments (exponentially many), and check if any of them satisfy the input formula. However, sometimes it is not necessary to assign all variables to detect that an assignment cannot satisfy the formula. For example the following formula cannot be satisfied by the assignment $b \leftarrow true$ and $c \leftarrow false$, regardless of the value of a or d:

$$(a \vee b) \wedge (\neg b \vee c) \wedge (a \vee \neg c \vee d)$$

Furthermore, an assignment can entail truth values for the unassigned variables. In the above example the assignment $c \leftarrow false$ entails $b \leftarrow false$.

The DPLL algorithm exploits this insight. DPLL is a backtracking search algorithm: at each step it guesses the value of a SAT variable in the input problem. It then explores the logical consequences of this decision by assigning variables to their entailed values. If no inconsistency is found and not all variables are assigned, another guess is made. If an inconsistency is found, the algorithm backtracks and tries to flip the truth value of the most recent guess. If it has been already flipped, the algorithm backtracks to the previous guess. The process continues until either a full assignment is found or no more backtracking is possible.

Next we will define some of the terms required to give a more precise description of DPLL. We will refer to the clauses in the input CNF-formula C as the *problem clauses*. The set of variables in a propositional formula ψ will be denoted by $vars(\psi)$.

An *ψ-assignment* A for a propositional formula ψ is a partial map from the SAT variables $vars(\psi)$ to the Boolean truth values $\{true, false\}$. We say a ψ-assignment is *complete* if all variables in $vars(\psi)$ are assigned. Otherwise it is *partial*. A C-assignment

A for a propositional CNF formula C is *inconsistent* if A falsifies a clause $c \in \mathsf{C}$. it is *consistent* otherwise.

The assignment is represented using a *trail*: a sequence of literals, where variables occurring in literals with a positive polarity are assigned to *true* and those with a negative polarity to *false*. With abuse of notation we will use A to denote both the trail and the map it represents.

For a given assignment A, a clause is *satisfied* if at least one of the literals it contains is assigned to *true* and it is *falsified* if all its literals are assigned to *false*. A clause is *unit under assignment* A if it has only one unassigned literal, and all the other literals are assigned to *false*.

Note that if a clause is unit under assignment A, the assignment along with the clause entail that the unassigned literal is *true*. This kind of reasoning is known as *unit propagation* and it is at the core of the entailment check in DPLL. *Boolean constraint propagation* (BCP) refers to applying unit propagation until no more literals can be propagated or a contradiction is reached (both x and $\neg x$ are propagated).

The guessed variables in the DPLL algorithm are called *decision* variables and we will mark them with superscript $(\_)^d$. Each literal l has a *decision level* $\mathsf{level}(l)$ defined as the number of decision literals occurring before it in the trail (including the literal itself). The total number of decision variables in a trail is the *current decision level*. The algorithms below explicitly store the current decision level in the dl variable. The level of a literal l is the value of dl when l is added to the trail.

Algorithm 1 shows the pseudo-code for DPLL. All procedures take their argument by reference and hence can change their values. The trail is stored in A and we use $[]$ for the empty trail and "::" for adding a new literal to the end of the trail. The decide procedure returns a new decision: it heuristically picks a literal that is not currently in the trail. The BCP procedure does Boolean-constraint propagation by adding the entailed literals to A. If it detects an inconsistency it returns a *conflict clause* c such that all the literals in c are assigned to *false* by the current trail.

BCP returns undef if no inconsistency is detected. The flipDecision procedure looks for the most recent decision that has not been flipped. It returns the backtracking level corresponding to this variable along with the literal representing the untried polarity. It returns -1 if the first decision variable has been tried in both polarities. The backtrack procedure pops the trail until it only contains literals at levels at most equal to the backtrack level b.

Algorithm 1: DPLL

```
Input: $\psi$ input formula
C ← toCnf($\psi$);
$\langle$A, dl$\rangle$ ← $\langle [], 0 \rangle$ ;
while true do
    c ← BCP(C, A) ;
    if c $\neq$ undef then
        $\langle b, l \rangle$ ← flipDecision(A) ;
        if $b = -1$ then
            return unsat;
        backtrack($b$, A) ;
        A ← A :: $\neg l$ ;
        continue;
    if allAssigned(A) then
        return sat;
    dl ← dl + 1;
    $l'$ ← decide();
    A ← A :: $l'$;
```

1.4.2 CDCL

One can think of the DPLL algorithm as "optimistically" trying to building a satisfying assignment. An alternative approach is to try to construct a proof that no satisfying assignment exist. This can be done using the *Boolean resolution rule*:

$$\frac{v \lor l_1 \lor \ldots \lor l_n \quad \neg v \lor l'_1 \lor \ldots \lor l'_n}{{}_1 \lor \ldots \lor l_n \lor l'_1 \lor \ldots \lor l'_n}\ \mathsf{Res}(,)$$

If the same SAT variable v occurs in two different clauses, positive in one and negative in the other, the rule infers a new clause, containing the union of the literals in the two clauses, with the exception of v and $\neg v$. Not only is this rule sound (the new clause is logically entailed by the two clauses), but it is also refutationally complete [76].

The CDCL algorithm extends DPLL with learning and non-chronological backtracking [10]. It does so by alternating trying to build a satisfying assignment via decisions with using resolution to learn new clauses that prune unfeasible parts of the search space.

Algorithm 2 shows the pseudo-code for CDCL. The highlighted lines are the lines that differ from DPLL. At the heart of CDCL is the analyzeConflict conflict analysis procedure which exploits the structure of the propagations to: (i) learn a new clause L entailed by the problem clauses and (ii) compute a backtracking level b, potentially lower than the most recent un-flipped decision. The learned clause L is added to the set of working clauses at line 7.

We will explain clause learning in CDCL using the notion of an *implication graph*. For each non-decision literal l let the *reason* clause $\mathsf{reason}(l)$ be the clause that led to l being propagated. Note that $l \in \mathsf{reason}(l)$ by the definition of BCP.

Definition 6. Given an inconsistent assignment[6] A and a set of clauses C the *implication*

---

[6] An implication graph can be defined for a consistent assignment by just dropping κ.

Algorithm 2: CDCL

```
Input: $\psi$ input formula
C ← toCnf($\psi$);
$\langle$A, dl$\rangle$ ← $\langle [], 0 \rangle$ ;
while true do
    c ← BCP(C, A);
    if c $\neq$ undef then
        $\langle b, L \rangle$ ← analyzeConflict(c) ;
        C ← C $\cup$ $L$;
        if $b = -1$ then
            return unsat;
        backtrack($b$, A);
        continue;
    if allAssigned(A) then
        return sat;
    dl ← dl + 1;
    $l$ ← decide();
    A ← A :: $l'$;
```

graph is $G = (V, E)$ where:

$$
\begin{aligned}
V &= \{l | l \text{ is a literal in } \mathsf{A}\} \cup \{\kappa\} \\
E &= \{(l_1, l_2) | \neg l_1 \in \mathsf{reason}(l_2), \mathsf{A}(\neg l_1) = false \text{ and } \mathsf{A}(l_2) = true\}
\end{aligned}
$$

The set V of vertices consists of literals as well as a specially designated vertex κ representing the conflict clause. Vertices without antecedents are decision literals or unit clauses.

Level	1	2		3	4				
Trail	x_1^d	$\neg x_2^d$	$\neg x_6$	$\neg x_5^d$	x_3^d	x_7	x_4	$\neg x_8$	x_8
Reason	$\bot$	$\bot$	c_1	$\bot$	$\bot$	c_2	c_3	c_4	c_5

Figure 1.1: Trail for Example 7.

Example 7. Consider the following set of input clauses C:

$$\begin{array}{rrrr} c_1: & \neg x_1 & x_2 & \neg x_6 \\ c_2: & x_6 & \neg x_3 & x_7 \\ c_3: & \neg x_3 & x_4 & \neg x_7 \\ c_4: & \neg x_4 & \neg x_8 & \\ c_5: & x_8 & \neg x_4 & x_5 \end{array}$$

with the corresponding trail and reason clauses shown in Figure 1.1. The first decided literal is x_1 at decision level 1, then $\neg x_2$ at level 2. Due to clause c_1 the literal $\neg x_6$ is now propagated at level 2 and so forth. Figure 1.2 shows the implication graph. Each node corresponds to a literal on the trail, and the number in parentheses is its decision level. Edges are labeled with the clause leading to the propagation.

The analyzeConflict procedure learns a new clause by finding a *unique implication point* (UIP): a vertex u in the implication graph such that any path from the current decision level of the graph to the conflict node κ goes through u. Intuitively, the UIP is a core reason for the conflict. In Example 7 the UIP is x_4.

An implication graph can contain more than one UIP but it has at least one (the current decision variable is always a UIP). Most modern CDCL SAT solvers use the first UIP from the conflict to determine the backtracking level and the learned clause. Algorithm 3 shows the pseudo-code for analyzeConflict. The procedure starts with the falsified conflict clause c and works backwards by resolving out literals until the first UIP

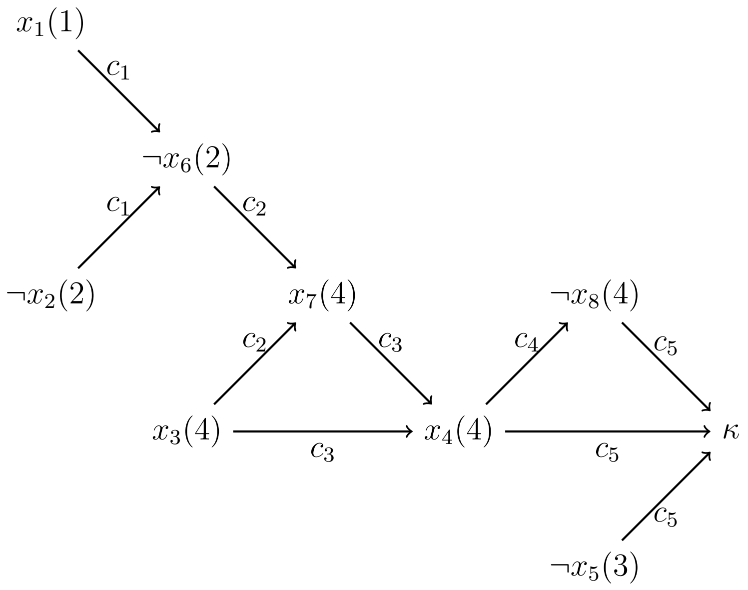

Figure 1.2: Implication graph for Example 7.

is found. The trail A can be viewed as a vector where $index_{\mathsf{A}}(l)$ returns the index of literal l. The seen set keeps track of literals already processed. The algorithm traverses the trail backwards: at each iteration the unprocessed literal in learned with the highest $index_{\mathsf{A}}$ value is resolved out.

Algorithm 3: The analyzeConflict procedure.

```
Input: $c$ conflict clause
learned $\leftarrow c$ ;
seen $\leftarrow \emptyset$;
while not hasUIP(learned) do
    $lit \leftarrow \arg\max_l \{index_{\mathsf{A}}(l) | l \in \text{learned} \setminus \text{seen}\}$;
    seen $\leftarrow$ seen $\cup \{lit\}$;
    if reason$(lit) \neq \bot$ then
        learned $\leftarrow$ Res(learned, reason$(lit)$);
bl $\leftarrow \max\{d | l \in \text{learned}, d = \text{level}(l) \text{ and } d \neq \text{dl}\}$ ;
return $\langle$bl, learned$\rangle$ ;
```

The hasUIP function checks if learned contains a UIP. It does so by checking whether there is only one literal l in $\mathsf{learned}$ such that $\mathsf{level}(l) = \mathsf{dl}$. Because the learned clause is obtained via resolution, it is entailed by the problem clauses. Furthermore, it is an *asserting* clause: after backtracking to bl it will be unit under the current trail and propagate the UIP in the opposite polarity.

In Example 7 the learned clause $L : \neg x_4 \lor x_5$ is obtained by resolving c_5 with c_4. The backtracking level is 3. After backtracking, x_5 is still assigned to *true*, so $\neg x_4$ is propagated: the UIP has flipped.

1.4.3 CDCL with assumptions

The CDCL algorithm can easily be enhanced to support limited incrementality. As a motivating example consider checking whether certain combination of inputs/outputs are valid for a circuit. The circuit can be modeled as a formula ψ and the fixed input/outputs as a set of literals assump called the *assumption literals*. We can then query the SAT solver for the satisfiability of $\psi \land \mathsf{assump}$ for each set of assumptions assump. We want to reuse some of the work of previous queries and not start from scratch for each assumption set. Asserting $\psi \land \mathsf{assump}$ as an input prevents us from reusing the learned clauses: these may not be implied by the next set of assumptions $\psi \land \mathsf{assump}'$. Forcing the literals in assump to be the first decisions the algorithm ensures that all learned clauses are still entailed by the problem clauses and can be reused between queries.

This technique is called *solve with assumptions* and was first proposed in [36]. The highlighted lines in Algorithm 4 show the changes needed to support assumptions. The $\mathsf{hasAssumps}$ method checks whether there are still assumptions to “decide”. If not, a regular decision is made. Otherwise, $\mathsf{nextAssump}$ gets the next undecided assumption literal l. If l is already assigned to *false*, the problem is unsatisfiable. Because no decisions were made yet, all the literals on the trail are entailed by the problem clauses and the assumptions processed so far. If l is already assigned to *true* (it was entailed by previous assumptions), we push a fake decision level.[7] This maintains the invariant that each assumption introduces a new decision level after it has been processed, although it is not necessarily the variable's decision level (it may have been propagated at a lower level). Note that with this invariant $\mathsf{hasAssumps}$ can be implemented efficiently by only checking whether dl is less than the number of literals in assump. We will later leverage this technique for solving bit-vector constraints in Section 4.2.

1.5 CDCL(T)

The propositional satisfiability algorithms described in Section 1.4 can be generalized to decide the satisfiability of first-order formulas with respect to a background theory by integrating one or more theory-specific solvers in the DPLL($\mathcal{T}$) framework [71]. The framework is named after the Davis-Putnam-Logemann-Loveland (DPLL) decision procedure for SAT. It extends propositional reasoning to reasoning in a theory $\mathcal{T}$ by relying on a *theory solver* ($\mathcal{T}$-solver): a decision procedure for the $\mathcal{T}$-satisfiability of $\mathcal{T}$-constraints (conjunctions of $\mathcal{T}$-literals).

Modern SMT solvers (including CVC4) have been taking advantage of the recent advances in SAT, and actually extend the CDCL SAT decision procedure. For this reason, we will refer to this framework as CDCL($\mathcal{T}$), although its historical name in the literature is DPLL($\mathcal{T}$).

For the rest of the thesis we will focus on the problem of deciding a *quantifier-free*[8] $\mathcal{T}$-formula for a single theory $\mathcal{T}$. The framework can be generalized to multiple disjoint

---

[7] For this to be consistent with the definition of decision level, we can think of making a ghost decision: deciding a fresh variable that does not appear in the problem.

[8] While SMT techniques for dealing with quantified formulas exist, they are outside the scope of this thesis.

Algorithm 4: CDCL with assumptions.

```
Input: $\langle\psi$, assump$\rangle$ input formula and assumptions
C ← toCnf($\psi$);
$\langle$A, dl$\rangle$ ← $\langle[], 0\rangle$ ;
while true do
    c ← BCP(C, A);
    if c $\neq$ undef then
        $\langle b, L\rangle$ ← analyzeConflict(c) ;
        C ← C $\cup$ $L$;
        if $b = -1$ then
            return unsat;
        backtrack($b$, A);
        continue;
    if allAssigned(A) then
        return sat;
    if hasAssumps(A, assump) then
        $l$ ← nextAssump(A, assump) ;
        if A($l$) = $false$ then
            return unsat;
        if A($l$) = $true$ then
            dl ← dl + 1;
    else
        $l$ ← decide();
    dl ← dl + 1;
    A ← A :: $l$;
```

theories by explicitly incorporating a Nelson-Oppen style combination of the individual theory solvers [52, 69].

Given $\mathcal{T}$ we denote by $(\_)^{\mathrm{P}}$ an arbitrary but fixed injective mapping from $\mathcal{T}$-atoms to propositional variables, and use $(\_)^{\mathrm{P}}$ also for its homomorphic extension to quantifier-free $\mathcal{T}$-formulas. $(\_)^{\mathrm{P}}$ provides a *propositional abstraction* of such formulas. We will use $\models_P$ to denote propositional satisfiability.

Given an arbitrary quantifier-free $\mathcal{T}$-formula ψ a CDCL-style SAT solver performs a search for a propositional assignment that satisfies the propositional abstraction ψ^{P} of ψ. The $\mathcal{T}$-solver decides the $\mathcal{T}$-satisfiability of the corresponding $\mathcal{T}$-constraint obtained by instantiating each variable in the assignment with the $\mathcal{T}$-atom it abstracts. If the constraint is not $\mathcal{T}$-satisfiable, a conflict clause consisting of a subset of the negated constraints is added to the SAT solver, forcing it to try a different assignment. The process is repeated until either a $\mathcal{T}$-satisfiable assignment is found or all propositional assignments are exhausted and the formula is unsatisfiable. While for completeness the $\mathcal{T}$-solver is only required to decide the $\mathcal{T}$-satisfiability of a conjunction of literals, for efficiency a closer coupling of the theory and the SAT solver is desirable.

Algorithm 5 gives a simplified algorithmic view of the CDCL($\mathcal{T}$) framework as it is implemented in the CVC4 SMT solver. The algorithm takes as an input a $\mathcal{T}$-formula ψ and returns sat if ψ is satisfiable and $unsat$ otherwise. For simplicity, the $\_^{\mathrm{P}}$ mapping will be implicit: we assume that $\mathcal{T}$-atoms are treated as propositional variables by the SAT solver and are converted back to $\mathcal{T}$-atoms when communicating with the theory via $\mathcal{T}$-propagate and $\mathcal{T}$-check.

The highlighted lines are those that differ from CDCL. After BCP, the $\mathcal{T}$-solver can add to the trail $\mathcal{T}$-literals that are $\mathcal{T}$-entailed by the current assignment ($\mathcal{T}$-propagate). This prevents making unnecessary, potentially costly decisions. Like BCP $\mathcal{T}$-propagate

Algorithm 5: CDCL($\mathcal{T}$)

```
Input: $\psi$ input formula
C ← toCnf($\psi$);
$\langle$A, dl$\rangle$ ← $\langle [], 0 \rangle$ ;
while true do
    c ← BCP(C, A);
    if c = undef then
        c ← $\mathcal{T}$-propagate(A)
    if c = undef then
        final ← satisfies($\psi$, A);
        c ← $\mathcal{T}$-check(A, final);
        if c ≠ undef then
            C ← C ∪ c;
            if not falsified(c) then continue;
    if c ≠ undef then
        $\langle b, L \rangle$ ← analyzeConflict(c) ;
        C ← C ∪ $L$;
        if $b = -1$ then
            return unsat;
        backtrack($b$);
        continue;
    if final then
        return sat;
    dl ← dl + 1;
    $l$ ← decideRelevant();
    A ← A :: $l$;
```

returns a conflict clause c if propagation detected a $\mathcal{T}$-inconsistency, and undef if no such inconsistency was found.[9]

Example 8. Consider the following set of $\mathcal{T}_{\mathsf{bv}}$ clauses:

$$\begin{array}{rccc} c_1: & x = y & b = 0_{[n]} & \\ c_2: & \neg x = y & \neg a = 0_{[n]} & z = x + y \\ c_3: & z = 7_{[n]} & x = \sim y & \end{array}$$

with the following trail $\mathsf{A} = [(x = y)^d, (a = 0_{[n]})^d, z = x + y]$. If $x = y$ then $z = 2 \times x$, so as a multiple of 2 it cannot be odd: $\neg z = 7_{[n]}$. $\mathcal{T}$-propagate can propagate this fact and add it to the trail avoiding a potentially wrong decision: $\mathsf{A} = [(x = y)^d, (a = 0_{[n]})^d, z = x + y, \neg z = 7_{[n]}]$

Recall that during BCP, each propagated literal is assigned a reason clause, that is used in analyzeConflict to learn the conflict clause. The same is required for $\mathcal{T}$-propagations. The theory solver must be able to provide an *explanation* (a $\mathcal{T}$-reason) for each theory-propagated literal l. This is a $\mathcal{T}$-valid clause of the form $\neg l_1 \vee \cdots \vee \neg l_n \vee l$ for some subset $\{l_1, \ldots, l_n\}$ of A explaining why the literal was entailed. In practice, a small subset of all propagations is involved in a conflict and requires explanations. For this reason, it is important that theory solvers compute explanations lazily, only as needed by analyzeConflict.

If no conflict is detected by $\mathcal{T}$-propagate, the $\mathcal{T}$-solver is asked to check the consistency of the current partial assignment.

Example 9. Continuing Example 8, BCP detects that $x = \sim y$ must be *true* to satisfy

---

[9] This does not necessarily mean the assignment is not $\mathcal{T}$-inconsistent.

clause c_3 and adds it to the trail:

$$\mathsf{A} = [(x = y)^d, (a = 0_{[n]})^d, z = x + y; \neg z = 7_{[n]}, x =\sim y].$$

The $\mathcal{T}$-check procedure now detects an inconsistency and returns a conflict clause

$$c_4 : \neg x = y \vee \neg x =\sim y$$

which is added to the set of clauses. The analyzeConflict procedure must learn a new clause. It first resolves out $\neg x =\sim y$ with reason$(x =\sim y) : c_3$ and obtains clause $\neg x = y \vee z = 7_{[n]}$. To resolve out the next literal $z = 7_{[n]}$ the $\mathcal{T}$-solver must be able to provide an explanation for $\neg z = 7_{[n]}$:

$$\mathit{explain}(\neg z = 7_{[n]}) : x = y \wedge z = x + y$$

The explanation is added as a clause to C and conflict resolution continues until a UIP is reached:

$$c_6 : \neg x = y \vee \neg z = x + y \vee \neg z = 7_{[n]}.$$

The final variable indicates whether the current partial assignment propositionally satisfies the original input formula ψ:[10]

Definition 7. An assignment A *propositionally satisfies* a quantifier-free formula ψ if $A^{\mathrm{P}} \models_P \forall \mathbf{v} \psi^{\mathrm{P}}$, where $\mathbf{v} = \{v \mid v \in \mathit{vars}(\psi^{\mathrm{P}}),\ v \notin \mathit{vars}(A^{\mathrm{P}})\}$.

Note that a propositionally satisfying assignment for a formula ψ need not be *full*, i.e., defined for all variables of ψ^{P}, but it can be extended arbitrarily to a full one. An

---

[10] Note that this is the formula before CNF conversion.

efficient implementation of satisfies is described in Section 2.3.

We say a call to $\mathcal{T}$-check is final if A propositionally satisfies the input formula (i.e. final is set to true). Final calls to $\mathcal{T}$-check must either ensure that A is $\mathcal{T}$-satisfiable, return a conflict clause, or add one or more theory lemmas:

1. Return undef if A is $\mathcal{T}$-satisfiable.
2. Return a $\mathcal{T}$-inconsistent subset of the literals in A in the form of a clause if A is $\mathcal{T}$-unsatisfiable.
3. If $\mathcal{T}$-check cannot efficiently determine the satisfiability of A, it may request a split by returning a new clause c that is $\mathcal{T}$-valid.

The third case above corresponds to the splitting on demand approach [8] that allows theories to delegate internal splitting to the SAT solver.

If final is not *true*, the $\mathcal{T}$-solver is allowed to do as much or as little work as it wants. Because both the conflict and the lemma are $\mathcal{T}$-valid, it is safe to add them to the list of working clauses. If the $\mathcal{T}$-check returned clause is not falsified by the current assignment, we skip making a decision and propagate again.

The decideRelevant procedure picks an unassigned literal that is relevant to the satisfiability of the input formula. The implementation of this procedure is related to that of satisfies and is described in Section 2.3.

An important aspect of theory solvers is not captured in the $\mathcal{T}$-interface of described in Algorithm 5. Actual implementations of $\mathcal{T}$-check and $\mathcal{T}$-propagate are stateful: they store a copy of the assignment A internally and are instructed to push and pop literals from it as A is modified by the main loop. In practice, it is crucial that the theory solver be able to backtrack efficiently when A is shrunk, and reason incrementally when it is extended.

In Chapter 4 we will use this framework to build a lazy $\mathcal{T}$-solver for the fixed-width bit-vector theory.

Chapter 2

CVC4

The bit-vector solvers presented in this thesis have been implemented in the general purpose SMT solver CVC4. This chapter gives an overview of CVC4's architecture while focusing on some of the core infrastructure that is particularly beneficial for solving bit-vector constraints. Most of the features described in this chapter are shared across multiple theories and have been implemented by various members of the CVC4 team. Only the bit-vector specific rewrites and simplifications are contributions of this thesis.

CVC4 implements the CDCL framework described in Section 1.4 and has support for several common first-order theories, including: bit-vectors, arrays, integer and real linear arithmetic, inductive data-types, uninterpreted function symbols, and strings. It is an open source software project: the source code is distributed under the Modified BSD license. Source code and pre-compiled binaries for several platforms can be downloaded from `cvc4.cs.nyu.edu`. For more details on CVC4 see [6].

Figure 2.1 shows a simplified view of CVC4's architecture. The input formula ψ is first preprocessed. Preprocessing consists of a series of equisatisfiable formula transformations that serve several purposes including: simplifying the formula, learning new

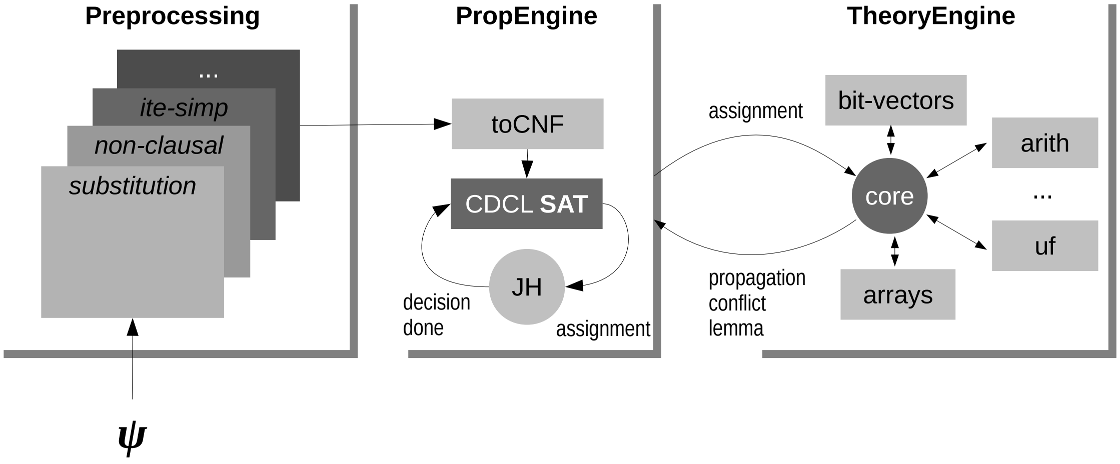

Figure 2.1: CVC4 architecture.

facts and putting the formula in a standard form that makes it easier for the rest of the system to reason about.

The preprocessed formula is then passed to the `propEngine` that does most of the propositional reasoning and maintains the map between the formula and its propositional abstraction. The Boolean abstraction is converted to CNF and asserted to the CDCL-style SAT solver. Optionally, the SAT solver can query a *justification heuristic* ($\mathrm{SAT}_{\mathcal{J}}$) engine (Section 2.3) to check whether the current partial assignment satisfies the formula and help it guide the search to relevant parts of the formula.

If no conflict is found, the truth assignment is communicated to the `theoryEngine`. The `theoryEngine` encompasses all theory reasoning. Its core component communicates the truth assignment to the relevant theories. If more than one theory is involved it also ensures that the theories agree on equalities over shared terms (*theory combination*). The individual theory solvers can identify $\mathcal{T}$-conflicts in the assignment, propagate $\mathcal{T}$-entailed literals and issue $\mathcal{T}$-valid lemmas.

The formulas manipulated by the procedures described above are represented as directed-acyclic graphs (DAGs) using the `Node` data-structure. The leaves of the DAG are either variables or constants. The inner nodes are operator applications and the children the operator arguments. This data-structure uses hash-consing to ensure that two structurally identical DAGs are represented by the same data-structure. As Nodes are used extensively throughout the system, an efficient implementation of this data-structure is key for performance.

2.1 Rewriting

CVC4 has a general rewriting module Rewriter that employs $\mathcal{T}$-valid rewrite rules to simplify terms and formulas. Its use is not limited to preprocessing operations: it is used throughout the system to normalize and simplify expressions. For this reason care must be taken to ensure that the rewrite rules are applied efficiently.

Each theory $\mathcal{T}$ implements a $\mathcal{T}$-Rewriter that operates on $\mathcal{T}$-literals and $\mathcal{T}$-terms. As $\mathcal{T}$-rewrites can be applied during search, the $\mathcal{T}$-Rewriter is not allowed to introduce Boolean structure and it must rewrite equalities to other equalities or to a Boolean constant.

Boolean simplifications are delegated to a special Boolean Rewriter. We will use $\mathsf{Rewrite}(t)$ to denote the term the Rewriter rewrites t to and $\equiv$ to denote syntactical equivalence. The CVC4 Rewriter has the following invariants:

1. Rewrites are theory-valid:
$$\models_{\mathcal{T}} t = \mathsf{Rewrite}(t)$$

2. Rewriting is idempotent:
$$\mathsf{Rewrite}(\mathsf{Rewrite}(t)) \equiv \mathsf{Rewrite}(t)$$

3. The $\mathsf{Rewriter}$ is *strongly normalizing* for terms over constants and interpreted function symbols. For all terms s and t built from interpreted constant symbols and interpreted function symbols:

$$\models_{\mathcal{T}} t = s \quad \text{iff} \quad \mathsf{Rewrite}(t) \equiv \mathsf{Rewrite}(s)$$

   Intuitively, this requirement states that the $\mathsf{Rewriter}$ evaluates constant terms.

While for certain theories the $\mathsf{Rewriter}$ can be strongly-normalizing over arbitrary terms, this is computationally intractable for theories like the bit-vector theory.

The CVC4 $\mathsf{Rewriter}$ consists of two passes over the DAG: a *pre-rewriting* pass where a node is visited before its children and a *post-rewriting* pass where a node is visited after all of its children have been rewritten. The pre-rewriting phase can eliminate potentially large expressions before even rewriting them. For example, if the $t_{[n]}$ term only occurs in the following sub-expression, the $\mathsf{Rewriter}$ can eliminate it without recursively rewriting $t_{[n]}$:

$$\mathsf{Rewrite}(0_{[n]} \& t_{[n]}) = 0_{[n]}$$

The post-rewriting phase takes advantage of the simplifications applied to children to simplify the parent Node and can assume the children have a specific form. For efficiency, the results of the pre and post-rewriting passes are cached.

Example 10. We use $\xRightarrow{\mathsf{Rewriter}}$ to show the intermediate steps in the rewriting process:

$$\begin{aligned}
(x_{[32]} + y_{[32]})[8:0] \leq (0_{[12]} \circ z_{[4]})[15:8] &\xRightarrow{\mathsf{Rewriter}} (x_{[32]} + y_{[32]})[8:0] \leq 0_{[8]} \\
&\xRightarrow{\mathsf{Rewriter}} x[8:0] + y[8:0] \leq 0_{[8]} \\
&\xRightarrow{\mathsf{Rewriter}} x[8:0] + y[8:0] = 0_{[8]}
\end{aligned}$$

No rewrite rule applies during pre-rewriting, so the children are rewritten first. The second child is simplified by pushing the extract operation over concatenation, and the first one by pushing the other extract over addition. This decreases the size of the bit-blasted addition circuit from 32 bits to 8. During post-rewriting, the inequality is rewritten to an equality: as this is unsigned comparison, the only way the sum can be less than or equal to 0 is if it is equal to 0.

2.2 Preprocessing

CVC4 has a sophisticated preprocessing module that heuristically combines several simplification passes. Some simplifications can only be applied soundly to formulas entailed by the input formula. We will say a $\mathcal{T}$-formula l is *top-level* w.r.t. an input formula ψ if $\psi \models_{\mathcal{T}} l$. A common case is that l is a conjunct of ψ: $\psi = \psi' \wedge l$. In this section we give a quick overview of the passes most useful for the bit-vector solvers:

Equality substitution. Top-level equalities of the form $v = t$ where v is a variable and t is an arbitrary term not containing v are used to substitute v for t in the entire formula:

$$\models_{\mathcal{T}} (\psi \wedge x = t) \Leftrightarrow \psi[v \leftarrow t]$$

where $\psi[v \leftarrow t]$ is short-hand for substituting v by t in formula ψ.[1] This is especially useful for the bit-vector theory, as it reduces the size of the bit-blasted formula by removing the circuit corresponding to the substituted equalities.

Unconstrained simplification. This pass identifies *unconstrained terms* [22]: terms that are not forced to take a particular value. For example, assume the input for-

---

[1] As ψ is quantifier-free, substitution of bound variables under quantifiers is not an issue.

mula contains the following sub-term $v_{[32]} + t_{[32]}$, where v is a variable and t is an arbitrary bit-vector term. If this is the only occurrence of the variable v, the following equality can always be satisfied for any term $t'_{[32]}$ not containing v:

$$v_{[32]} + t_{[32]} = t'_{[32]}$$

As v is *unconstrained* we can always set it to $t'_{[32]} - t_{[32]}$. Therefore $v_{[32]} + t_{[32]}$ is also unconstrained, and can be replaced by a fresh variable. This is particularly useful for the bit-vector theory as potentially expensive circuits are removed from the problem. Note that the unconstrained property does not propagate through all operators. For example, even if v is unconstrained, $2_{[32]} \times v_{[32]}$ is not as its last bit must be 0.

Ite simplification. If-then-else terms implicitly contain a disjunction. They are usually eliminated by introducing a fresh variable for the ite-term. For each ite-term $ite(c, t_1, t_2)$ occurring in the input formula φ, ite-removal will introduce a fresh Skolem variable s_{ite}:

$$\begin{aligned}\varphi[ite(c, t_1, t_2)] \xRightarrow{\mathsf{RemoveIte}} \varphi[s_{ite}] \quad & \wedge(c \Rightarrow s_{ite} = t_1)\\ & \wedge(\neg c \Rightarrow s_{ite} = t_2).\end{aligned}$$

However, this can introduce redundancies. To counteract this effect, the CVC4 ite simplification pass is analogous to the procedure described in [54]. It applies ite simplifications and heuristic ite co-factoring to simplify the formula and improve the efficiency of the search. For example the following expression can be simplified to *false*:

$$\begin{aligned} 5_{[8]} = ite(c, ite(\neg c, t, 3_{[8]}), 7_{[8]}) &\xRightarrow{\mathsf{SimpIte}} 5_{[8]} = ite(c, 3_{[8]}, 7_{[8]}) \\ &\xRightarrow{\mathsf{SimpIte}} \bot \end{aligned}$$

Top-level simplifications. Certain simplifications may be sound or beneficial only if the formula they apply to is top-level. These simplifications usually introduce fresh Skolem variables or have the potential of increasing the size of the formula.

Example 11. One such top-level simplification is *variable slicing*. Slicing by introducing Skolem variables can enable new substitutions. Consider the equality:

$$v_{[32]}[15:8] = c_{[8]} \tag{2.1}$$

where $c_{[8]}$ is a constant, and v is a variable. This is equivalent to:

$$v_{[32]} = s_{[16]} \circ c_{[8]} \circ s'_{[8]} \tag{2.2}$$

for fresh Skolem variables s and s'. Replacing 2.1 by 2.2 enables substituting v by the right-hand-side and can potentially lead to further simplifications. However, it is not sound to apply this simplification if 2.1 is not top-level because s and s' are implicitly existentially quantified. Applying the simplification under a negation would result in the following universally quantified formula:

$$\begin{aligned} \neg(v_{[32]}[15:8] = c_{[8]}) &\Leftrightarrow \neg(\exists x, x'.v_{[32]} = x_{[16]} \circ c_{[8]} \circ x'_{[8]}) \\ &\Leftrightarrow \forall x, x'.v_{[32]} \neq x_{[16]} \circ c_{[8]} \circ x'_{[8]} \end{aligned}$$

This is not equivalent to $v_{[32]} \neq s_{[16]} \circ c_{[8]} \circ s'_{[8]}$.

Example 12. The following equality is a common way of encoding the fact that v is a power of 2:

$$v_{[n]} \& (v_{[n]} - 1_{[1]}) = 0_{[n]}$$

An equivalent way of expressing the same fact is that $v_{[n]}$ is the result of left-shifting $1_{[n]}$ by some $x < n$:

$$\exists x.(v_{[n]} = 1_{[n]} << x_{[n]}) \wedge x_{[n]} < n_{[n]}.$$

If the assertion is top-level, we can replace the existentially quantified formula by a fresh Skolem s as free-variables are implicitly existentially quantified:

$$(v_{[n]} = 1_{[n]} << s_{[n]}) \wedge s_{[n]} < n_{[n]}.$$

The second form enables a new substitution and restricts the search space of all possible values of v to n possible values as opposed to 2^n.

2.3 Justification heuristic

The justification heuristic engine (SAT$_{\mathcal{J}}$) relies on non-clausal reasoning to reduce the number of $\mathcal{T}$-atoms the $\mathcal{T}$-solvers have to reason about. It achieves this goal in two ways: (i) by identifying when a partial assignment A becomes propositionally satisfying (Definition 7) and (ii) by preventing the SAT solver from deciding literals that are not relevant in the current search context. Recall that traditional CDCL SAT solvers terminate and conclude that the input is satisfiable if all the variables have been assigned (allAssigned on line 12 in Algorithm 2, Section 1.4). If the input formula is purely propositional this is an efficient solution: keeping track of satisfied clauses is expensive while assigning an irrelevant variable is usually cheap. Once a partial assignment A satisfies a set of clauses C the SAT solver terminates with a linear amount of work in the size of the clause data-base. In SMT however, each assignment has to be checked for $\mathcal{T}$-consistency which can lead to potentially expensive calls to $\mathcal{T}$-solvers.

The non-clausal engine SAT$_{\mathcal{J}}$ employs circuit-based techniques of maintaining *justification frontiers*. Initially proposed in the context of automatic test pattern generation [43], these techniques have later been adapted to SMT [7].[2] The SAT$_{\mathcal{J}}$ engine keeps track of the relevant part of the formula by incrementally computing the *justification frontier* as the assignment A changes.

Example 13. Let the input formula be $\psi = \neg a \wedge (b \vee \varphi)$, and the current assignment $\mathsf{A} = [\neg a; b]$. SAT$_{\mathcal{J}}$ traverses the formula ψ and determines whether ψ is justified *true* by the assignment A. $\mathcal{T}$-atoms are *justified* to the values they are assigned to and unjustified otherwise. All other nodes are justified if their truth value is entailed by their justified children. In our example, to justify that the conjunction ψ is *true*, we need to justify both children as being *true*. The first conjunct is already justified as $\neg a$ is part of the assignment A. The second conjunct, $(b \vee \varphi)$, is *true* if at least one of b or φ is justified *true*. In our case b is *true* in the assignment, so the disjunction is also justified. The assignment of all other literals in φ is irrelevant.

Stopping the SAT search when a satisfying assignment is found does not prevent the SAT solver from deciding on literals that may not be relevant to the satisfiability of ψ. Instead of using the internal SAT solver heuristics, the SAT solver can rely on the non-clausal engine to pick the literals to branch on each time it needs to make a decision. Consider the formula ψ in the previous example, with the assignment $\mathsf{A} = [b]$.

---

[2] A different technique of reducing the number of literals sent to theory solvers is proposed in [31].

The clausal solver will query $\mathrm{SAT}_{\mathcal{J}}$ for a literal to decide on. As all the literals in φ are irrelevant, $\mathrm{SAT}_{\mathcal{J}}$ will return $\neg a$.

The strategy for combining the non-clausal engine $\mathrm{SAT}_{\mathcal{J}}$ with the clausal CDCL SAT solver in the CDCL($\mathcal{T}$) framework is shown in Algorithm 5 in Section 1.5. The justification heuristic engine is used in the procedure satisfies at line 8 to check whether the current partial assignment is satisfying and in decideRelevant at line 23 to force the SAT solver to decide a relevant literal.

The justification heuristic is usually enabled by default only for *expensive* theories such as bit-vectors and quantified formulas. The performance impact of the justification heuristic is detailed in Section 4.5. If the justification heuristic is disabled, satisfies and decideRelevant behave like allAssigned and the internal SAT solver heuristic decide respectively.

2.4 Bit-vector Solvers

CVC4 can be configured to use one of two different bit-vector solvers: an eager solver (cvcE) or a lazy solver (cvcLz). The two solvers take advantage of the same preprocessing techniques and rewrites, but employ different solving techniques.

The eager solver cvcE by-passes the CDCL($\mathcal{T}$) infrastructure, and encodes the entire $\mathcal{T}_{\mathsf{bv}}$-formula into propositional logic. It then relies on a second SAT solver to decide its satisfiability. See Chapter 3 for a detailed description of cvcE.

The lazy solver cvcLz, behaves like the other $\mathcal{T}$-solvers: the main SAT solver reasons about the Boolean abstraction of the input formula, and the $\mathcal{T}_{\mathsf{bv}}$-solver only reasons about conjunctions of $\mathcal{T}_{\mathsf{bv}}$ constraints. Chapter 4 describes the algorithms used by cvcLz. Finally Chapter 5 gives a detailed comparison of the performance of the solvers, analyzing the strengths and weaknesses of the two approaches.

Chapter 3

Eager Bit-vector Solving

Approaches to deciding the satisfiability of bit-vector formulas often rely on eager reduction to propositional logic and do not fit in the CDCL($\mathcal{T}$) framework of general-purpose SMT solvers. The properties of modular arithmetic combined with bit-level manipulations make algebraic reasoning in the bit-vector theory challenging. Reduction to SAT is appealing as it offers a unified way of reasoning about both the Boolean abstraction and the bit-vector constraints. This enables using the SAT solver as a black box and readily taking advantage of progress in SAT solving. We will call solvers that decide a $\mathcal{T}_{\mathsf{bv}}$-formula by bit-blasting the entire formula before starting the SAT search *eager bit-blasting solvers*. These solvers usually do employ algebraic reasoning, but only as a preprocessing step by first apply sophisticated word-level simplifications before bit-blasting.

In this chapter we described the implementation of the eager bit-vector solver cvcE in the SMT solver CVC4. Initially implemented as a base-line to evaluate the performance of the lazy solver cvcLz (see Chapter 4), the eager solver cvcE is now an essential option for tackling hard bit-vector problems.

Section 3.1 presents the overall architecture of the eager solver, and shows how it relates to that of traditional CDCL($\mathcal{T}$) theory solvers. Section 3.2 gives an overview of the bit-blasting module that converts the $\mathcal{T}_{\mathsf{bv}}$ input formula into a propositional logic formula. This module is very similar to the one used by the lazy solver for its bit-blasting. Section 3.3 describes the eager solver's use of the abc AIG rewriting module and its performance impact. Section 3.4 presents a method for reducing the size of the bit-blasted circuit by refactoring isomorphic circuits. We conclude with related work in Section 3.5.

3.1 Architecture

Given a quantifier-free $\mathcal{T}_{\mathsf{bv}}$ input formula ψ, cvcE decides its satisfiability by first applying the preprocessing techniques described in Section 2.2. The preprocessed formula is then converted to propositional logic by the bitblaster and then asserted to a CDCL-style SAT solver. Figure 3.1 shows the architecture of the eager bit-vector solver cvcE.

The preprocessing module is extended with a special pass, bvToBool, that attempts to lift bit-vector operations over bit-vectors of width 1 to operations over Booleans. For example, consider the formula:

$$\big(\sim (x[i:i]) \mathbin{\&} ite(c, x_{[1]}, 0_{[1]}) = 1_{[1]}\big) \land \varphi$$

where φ and c are arbitrary formulas, and x is an arbitrary bit-vector term. It is equivalent to the following:

$$\neg(x[i:i] = 1_{[1]}) \land ite(c, x_{[1]} = 1_{[1]}, \bot) \land \varphi.$$

Pushing some of the operations over bit-vectors of size 1 into the Boolean layer enables the other CVC4 preprocessing passes to exploit this Boolean structure to learn new top-level facts. In the above example, we can learn $x[i : i] = 0_{[1]} \wedge c$ after applying Boolean circuit propagation. This pass can also be helpful for the lazy bit-vector solver as we will see in Chapter 4.

The cvcE solver has very limited support for theory combination via *Ackermannization* [1]. This ackermannize pre-processing pass reduces constraints over the combination of $\mathcal{T}_{\mathsf{bv}}$ and $\mathcal{T}_{\mathsf{uf}}$ to $\mathcal{T}_{\mathsf{bv}}$ as follows:

- For each application $f(\bar{x})$ where $\bar{x} = (x_1, \dots, x_n)$, introduce a fresh variable $f_{\bar{x}}$ and use it to replace all occurrences of $f(\bar{x})$.
- For each $f(\bar{x})$ and $f(\bar{y})$ with $\bar{x} = (x_1, \dots, x_n)$ and $\bar{y} = (y_1, \dots, y_n)$ occurring in the input formula, add the following lemma:

$$\left(\bigwedge_{i=1}^{n} x_i = y_i\right) \Rightarrow f_{\bar{x}} = f_{\bar{y}}.$$

The pre-processed $\mathcal{T}_{\mathsf{bv}}$-formula is converted to propositional logic by the bitblaster. The bitblaster has two modes of operation: it can either bit-blast to an and-inverter-graph (AIG) or to an arbitrary Boolean formula. If the AIG path is enabled, the abc AIG rewriting package [20] is used to simplify the formula and convert it to CNF. If the Boolean path is enabled, the formula is bit-blasted to Nodes and CVC4's Boolean rewrites are applied. The bit-blasted formula is then encoded in CNF using a Tseitin-style conversion [81]. Finally, the CNF formula, obtained through either path, is asserted to the CDCL-style SAT solver $\mathrm{SAT}_{\mathsf{bb}}$. Our implementation is based on MiniSAT 2.2.0 [37, 77], but the framework allows for plugging in another SAT solver. Note that $\mathrm{SAT}_{\mathsf{bb}}$ is *not* the same SAT solver used to drive the search over the Boolean abstraction of the formula in CDCL($\mathcal{T}$). To distinguish between the two, we will refer to the CDCL($\mathcal{T}$) SAT solver as $\text{SAT}_{\mathsf{main}}$.

This design choice poses some interesting trade-offs. An alternative implementation could communicate the bit-blasted formula to $\text{SAT}_{\mathsf{main}}$ by asserting the bit-blasted clauses as $\mathcal{T}_{\mathsf{bv}}$-lemmas and adding them alongside the clauses that model the Boolean abstraction. In this scenario the $\mathcal{T}_{\mathsf{bv}}$ solver would be just a "shell" that pushes the bit-blasted formula to $\text{SAT}_{\mathsf{main}}$. An advantage of this approach is that it reuses existing infrastructure and allows for seamless theory combination: $\text{SAT}_{\mathsf{main}}$ already has the infrastructure to communicate with other involved theories and the $\mathcal{T}_{\mathsf{bv}}$ solver is reduced to a series of lemmas.

Our first prototype implementation used this idea. However, performance was significantly poorer. We attribute this to several causes. First, using a separate SAT solver for bit-vector reasoning allows for configuring the solver heuristic for bit-vector constraints. Second, $\text{SAT}_{\mathsf{main}}$ is limited in the number of SAT preprocessing techniques it can employ soundly: it is not usually sound to eliminate a $\mathcal{T}$-literal. Using SAT_{bb} allows for applying the MiniSAT preprocessing techniques, such as subsumption and variable elimination [35]. Care must be taken of course not to eliminate the bits of input variables if models are requested. Lastly, $\text{SAT}_{\mathsf{main}}$ requires non-trivial modifications to be integrated in CDCL($\mathcal{T}$). These modifications can add unnecessary overhead and restrict the functionality of the solver. Furthermore, plugging in a new CDCL($\mathcal{T}$) SAT solver requires a significant implementation effort, while trying a new SAT solver for SAT_{bb} requires implementing a minimal interface.

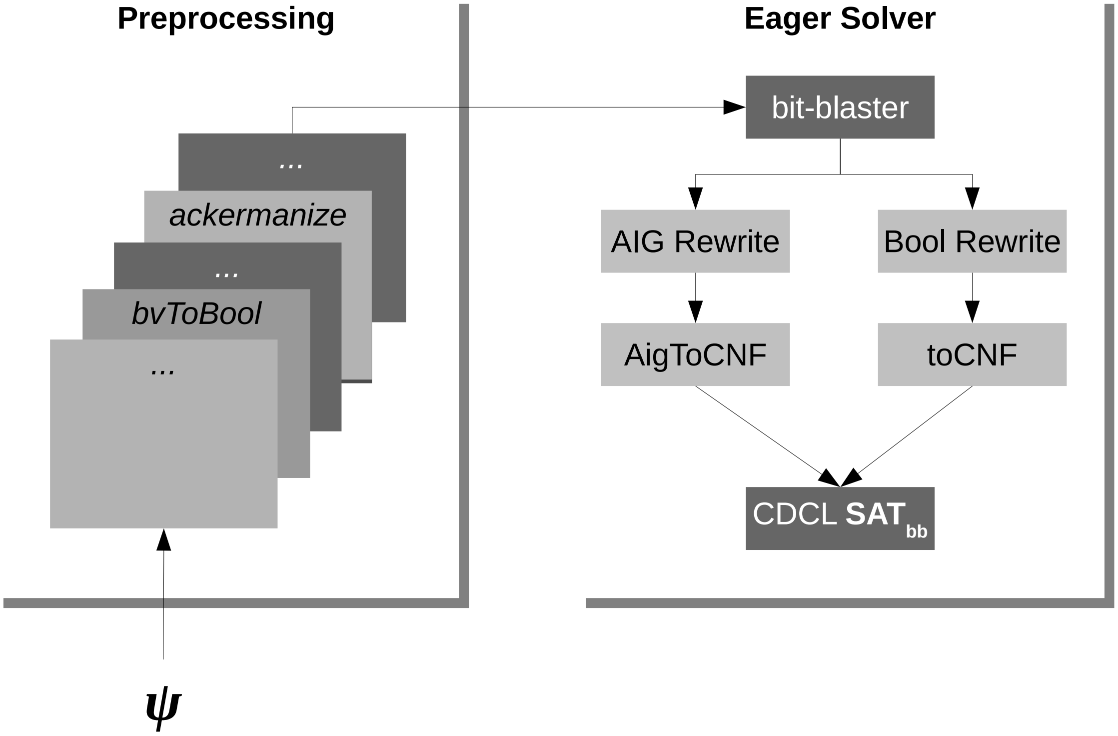

Figure 3.1: cvcE architecture.

3.2 Bit-blasting

Bit-blasting is the process of encoding bit-vector terms and formulas into propositional logic. In CVC4, bit-blasted terms are represented as a n-tuple of Boolean formulas, each representing the corresponding bit:

$$\mathsf{bbTerm}(t_{[n]}) \equiv \langle b_{n-1}, \ldots, b_0 \rangle$$

where b_i is the i^{th} bit of $t_{[n]}$. For a variable term, each bit is a fresh Boolean variable. For constant terms zero bits are *false* and one bits are *true*. Bit-blasting the bit-vector term operators proceeds recursively. Algorithm 6 shows the pseudo-code for bit-blasting the bit-wise and operator $\&$.

Bit-blasted atoms are represented as a Boolean formula: the formula corresponding

Algorithm 6: Bit-blasting bit-wise and.

```
Input: $t_{[n]} \& t'_{[n]}$
$\langle b_{n-1}, \ldots, b_0 \rangle \leftarrow \mathsf{bbTerm}(t_{[n]})$;
$\langle b'_{n-1}, \ldots, b'_0 \rangle \leftarrow \mathsf{bbTerm}(t'_{[n]})$;
return $\langle b_{n-1} \wedge b'_{n-1}, \ldots, b_0 \wedge b'_0 \rangle$;
```

Algorithm 7: Bit-blasting equality.

```
Input: $t_{[n]} = t'_{[n]}$
$\langle b_{n-1}, \ldots, b_0 \rangle \leftarrow \mathsf{bbTerm}(t_{[n]})$;
$\langle b'_{n-1}, \ldots, b'_0 \rangle \leftarrow \mathsf{bbTerm}(t'_{[n]})$;
return $\bigwedge_{i=0}^{n-1} b_i \Leftrightarrow b'_i$;
```

to atom a is $\mathsf{bbAtom}(a)$. They are built recursively from the bit-blasted terms. For example, equality is bit-blasted as shown in Algorithm 7. The final bit-blasted formula is obtained by asserting that the bit-blasted definition of each $\mathcal{T}_{\mathsf{bv}}$-atom is equivalent to the literal a occurring in the input formula:

$$\mathsf{bitblast}(\psi) \equiv \psi \wedge \bigwedge_{\text{atom } a \in \psi} (a \Leftrightarrow \mathsf{bbAtom}(a)) \,.$$

Bit-blasted terms and formulas are cached. Therefore, bit-blasting $x[i:0] \times y[i:0]$ after having already bit-blasted $x \times y$ should not add any new clauses. Because we represent bit-blasted terms as tuples of bits and do not explicitly introduce a fresh variable for each bit until CNF conversion, our bit-blasting procedure does not introduce fresh variables for *bit-propagating* operations (e.g. concatenate and extract) as described in [66]. The formulas corresponding to the atoms are simplified, either by calling the Boolean Rewriter or by using AIG rewriting, depending on which path is enabled.

3.3 AIG Rewriting

If the solver is configured to bit-blast to AIG the bit-blasted input formula is simplified using the combinational synthesis engine in abc [20].

And-inverter-graphs are a restricted form of Boolean formulas that only use the $\wedge$ and $\neg$ Boolean operators. An AIG is a DAG in which each node has either 2 or 0 children. Nodes with 2 children are $\wedge$-nodes, and nodes with no children are Boolean variables. Edges can be inverted or not: an inverted edge indicates that the child is negated. Such a restrictive representation has the advantage of making it easier to identify redundancies.

Our implementation uses by default the simplifications implemented by the abc command "*balance; rewrite*", but it can be configured to use user-provided abc scripts. The abc *balance* command minimizes the depth of the AIG by employing algebraic tree-balancing of the $\wedge$ nodes (inverted edges do not count towards the depth). The abc *rewrite* command implements an AIG rewriting algorithm that extends the work of [14]. For each 4-input cut of an AIG node, it checks a pre-computed table for equivalent ways to express the same Boolean function. The form that maximizes sharing in the overall DAG is chosen [64].

After applying the abc simplifications, we rely on abc's own CNF conversion algorithm to convert the AIG to clauses and assert them to SAT_{bb}. Their algorithm has been optimized for converting AIGs to CNF.

Experimental evaluation. While applying AIG rewriting to the entire bit-blasted formula can add a significant overhead for large instances, it proved to be highly beneficial for equivalence checking problems.

Experiments in this section were run on the StarExec [80] cluster infrastructure with

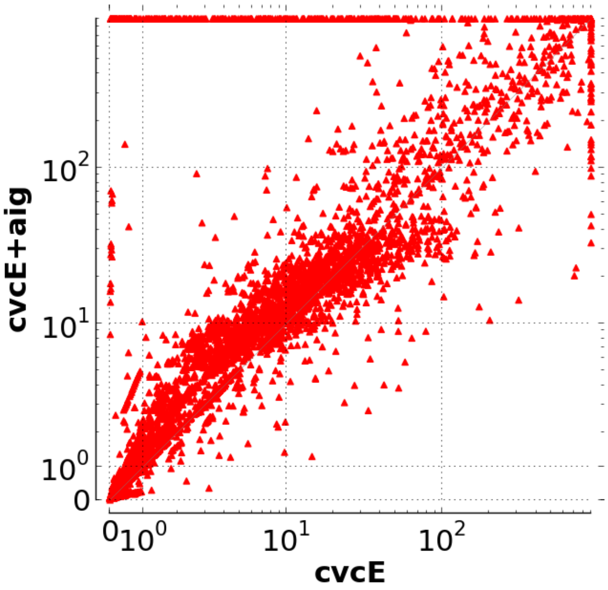

Figure 3.2: cvcE vs cvcE+AIG.

a timeout of 900 seconds and a memory limit of 50GB.[1] All scatter plots are on a log-scale, the x and y-axis represent CPU time in seconds, unless otherwise specified. Figure 3.2 compares the CVC4 eager solver performance by bit-blasting to AIG and applying AIG simplifications (cvcE+AIG) to bit-blasting to Node and applying Boolean rewrites (cvcE). The results are mixed, with the performance of cvcE+AIG being worse overall.

Focusing on specific families sheds more light on the problem as distinct patterns begin to emerge. Figure 3.3 shows the impact of AIG rewriting on the brummayerbiere* (brummayerbiere, brummayerbiere2, brummayerbiere3, and brummayerbiere4) SMT-LIB 2014-06-03 benchmark families. Most of the problems in these families encode equivalence checking problems adapted from [83]: low-level bit-fiddling operations are

---

[1] Experiments were ran on the queue all.q consisting of Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz machines with 268 GB of memory.

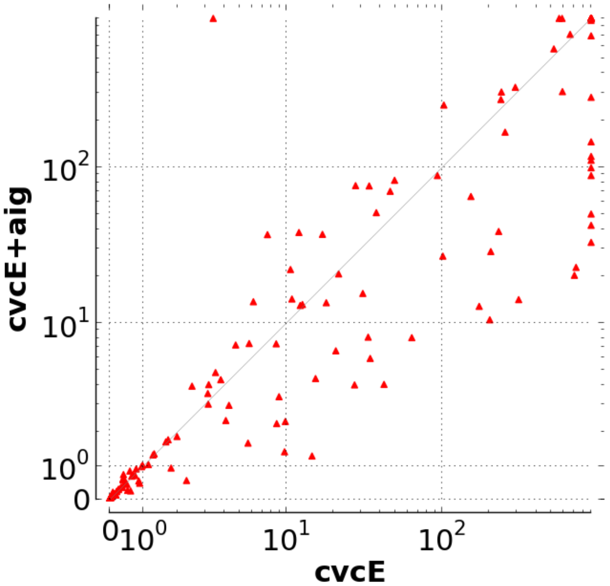

Figure 3.3: cvcE vs cvcE+AIG on brummayerbiere*.

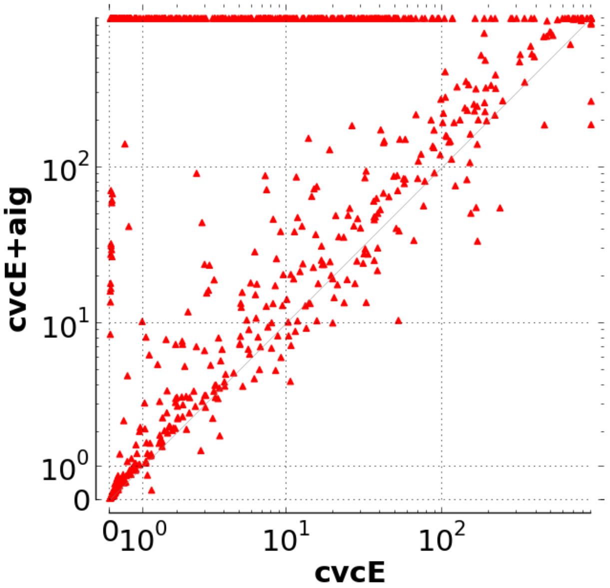

Figure 3.4: cvcE vs cvcE+AIG on bruttomesso.

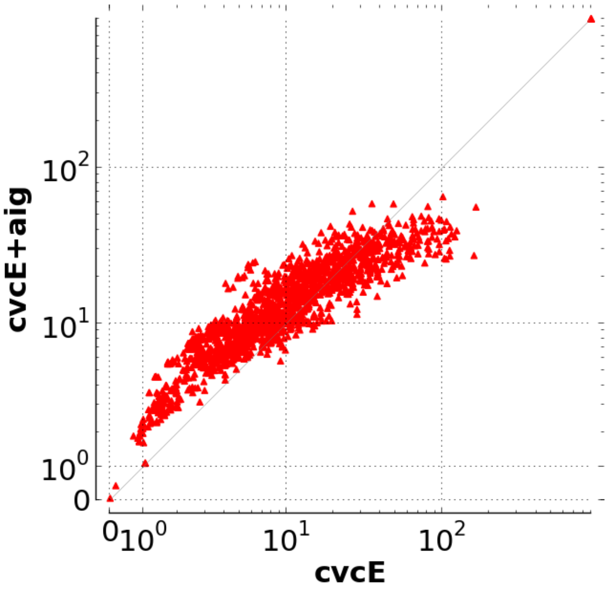

Figure 3.5: cvcE vs cvcE+AIG on spear.

checked for equivalence to their word-level specification. Removing redundancy via AIG rewriting is very helpful for these benchmarks.

On the other hand, AIG rewriting seems to have a very negative effect on problems that are very structured, such as the crafted bruttomesso family. Figure 3.4 shows this negative impact. On these problems, AIG rewriting does not take a long time, but in some cases bit-blasting to AIG increases the number of literals in the bit-blasted formula by 20% compared to bit-blasting to Nodes.

Figure 3.5 shows the performance on the larger industrial benchmark family spear (1695 benchmarks). The plot shows how on easier problems the overhead of AIG rewriting hurts performance while for harder problems it is outweighed by the reduction in solving time.

3.4 Refactoring Isomorphic Circuits

Problems arising from practical applications are often very structured and can sometimes exhibit repeated occurrences of the same pattern. To take advantage of this structure we developed a preprocessing pass aimed at identifying isomorphic formulas and factoring them out by introducing fresh variables. Consider the following constraint:

$$\left(\begin{array}{ll} x_0 = 2 \times y_0 + y_1 & \vee \\ x_0 = 2 \times y_1 + y_2 & \vee \\ x_0 = 2 \times y_2 + y_0 & \end{array}\right) \wedge \left(\begin{array}{ll} x_1 = 3 \times y_0 + 2 \times x_0 + 5 & \vee \\ x_1 = 3 \times y_1 + 2 \times x_0 + 5 & \vee \\ x_1 = 3 \times x_0 + 2 \times y_2 + 5 & \end{array}\right)$$

The disjuncts in each conjunct implement the same function applied to different arguments:

$$\left(\begin{array}{ll} x_0 = f(y_0, y_1) & \vee \\ x_0 = f(y_1, y_2) & \vee \\ x_0 = f(y_2, y_0) & \vee \end{array}\right) \wedge \left(\begin{array}{ll} x_1 = g(y_0, x_0) & \vee \\ x_1 = g(y_1, x_0) & \vee \\ x_1 = g(x_0, y_2) & \vee \end{array}\right)$$

where $f(x, x') = 2 \times x + x'$ and $g(x, x') = 3 \times x + 2 \times x' + 5$. During bit-blasting each application of f and g will be bit-blasted to a different but isomorphic set of clauses. Any clause learned for one of the function applications will not translate to the others. For example, in the $x_0 = 2 \times y_0 + y_1$ equality, x_0 and y_1 must have the same last bit, since the last bit of $2 \times y_0$ is 0. We would need to learn similar clauses encoding this fact, for each application of f:

$$\begin{aligned}
x_0[0:0] = 0 \vee y_1[0:0] = 1\\
x_0[0:0] = 0 \vee y_2[0:0] = 1\\
x_0[0:0] = 0 \vee y_0[0:0] = 1.
\end{aligned}$$

We can exploit this structure by introducing fresh variables for the arguments of f and g and pushing the disjunction to equalities over the arguments as follows:

$$\begin{array}{cc}
x_0 = f(s_0, s_1) \wedge & \left(\begin{array}{ll} (s_0 = y_0 \wedge s_1 = y_1) & \vee \\ (s_0 = y_1 \wedge s_1 = y_2) & \vee \\ (s_0 = y_2 \wedge s_1 = y_0) & \vee \end{array}\right) \wedge \\
x_1 = g(s'_0, s'_1) \wedge & \left(\begin{array}{ll} (s'_0 = y_0 \wedge s'_1 = x_0) & \vee \\ (s'_0 = y_1 \wedge s'_1 = x_0) & \vee \\ (s'_0 = x_0 \wedge s'_1 = y_2) & \vee \end{array}\right)
\end{array}$$

If the circuits of f and g are large enough this can lead to a significant reduction in the size of the bit-blasted circuit.

Algorithm 8 shows the pseudo-code for the refactoring isomorphic circuits pass. The pass first identifies top-level disjunctions φ and looks for a recurring pattern within the disjuncts. The patternOf procedure computes the pattern for each disjunct by replacing each variable v by a placeholder $\square_n$. For example:

$$\begin{aligned}
\mathsf{patternOf}(3 \times y + 2 \times x + 5) &= 3 \times \square_0 + 2 \times \square_1 + 5\\
\mathsf{patternOf}(x \times x + 5) &= \square_0 \times \square_1 + 5
\end{aligned}$$

Note that in the second term, the second occurrence of x had the same place-holder as the first. This is to minimize the number of argument equalities we need to add.

It may be the case that a pattern is a special case of another pattern. We say a pattern p' is a *generalization* of pattern p ($\mathsf{p}' \succeq \mathsf{p}$) if there is a substitution that can be applied to p' that makes p syntactically equal to p'. For example:

$$\begin{aligned} \square_0 + 5 &\preceq \square_0 + \square_1 \\ \square_0 + 5 &\not\preceq \square_0 + \square_0. \end{aligned}$$

Note that this is not the same as unification: the second example is unifiable but the second pattern is not a generalization of the first. The procedures getGeneralization and setGeneralization ensure that the patterns used for refactoring are the most general ones to maximize the number of disjuncts refactored.

The map args maps patterns to the arguments they need to be instantiated to. The $\mathsf{getArgs}(\varphi_i, \mathsf{p})$ procedure returns the arguments p needs to be instantiated with to refactor φ_i. Finally, skolemizeArgs introduces the disjunction of equalities and collapses disjuncts with the same pattern into one instantiation of the pattern. Note that each top-level disjunction φ can have more than one pattern that is refactored.

Experimental evaluation. This pass is very useful in reducing the size of the bit-blasted circuit when it applies, and adds negligible overhead when it does not. The most significant performance gain we observed was on the mcm family of benchmarks that exhibit this structure extensively. The mcm family encodes the multiple constant multiplication problem: synthesize an optimal sequence of operations that result in multiplying a given input by a fixed set of constants [62]. Figure 3.6 is a scatter plot comparing the size of the bit-blasted formula before applying circuit refactoring (cvcE) and after (cvcE+RIC). Each point is a benchmark and the x and y axis represent the thousands of

Algorithm 8: Refactoring isomorphic circuits.

```
Input: $\psi$
$\psi' \leftarrow \psi$;
for $\varphi \in \psi$ and $\varphi$ top-level do
    if $\varphi = \bigvee \varphi_i$ then
        $\mathsf{P} \leftarrow \emptyset$;
        for $\varphi_i \in \varphi$ do
            p ← patternOf($\varphi_i$);
            P ← P ∪{p};
        for $\mathsf{p}, \mathsf{p}' \in \mathsf{P}$ do
            if $\mathsf{p} \preceq \mathsf{p}'$ then
                setGeneralization(p, p');
        for $\varphi_i \in \varphi$ do
            p ← getGeneralization(patternOf($\varphi_i$));
            args(p) ← args(p) ∪ getArgs($\varphi_i$, p);
        $\varphi'$ ← skolemizeArgs($\varphi$, args);
        $\psi' \leftarrow \psi'[\varphi \leftarrow \varphi']$ ;
return $\psi'$;
```

literals in the bit-blasted formula.

Figure 3.7 shows the performance impact of the reducing the bit-blasted formula size. The x-axis is number of problems solved, and the y axis is time taken to solve that number of problems. The reduced circuit size results in one more problem solved and requires significantly less time.

3.5 Related Work

There are several eager solvers that are specialized for only reasoning about $\mathcal{T}_{\mathsf{bv}}$ formulas, usually in combination with the theory of arrays ($\mathcal{T}_{\mathsf{arr}}$) and the theory of uninterpreted function symbols ($\mathcal{T}_{\mathsf{uf}}$). For $\mathcal{T}_{\mathsf{bv}}$ formulas, these solvers rely on sophisticated word-level and bit-level rewrites and preprocessing techniques before encoding

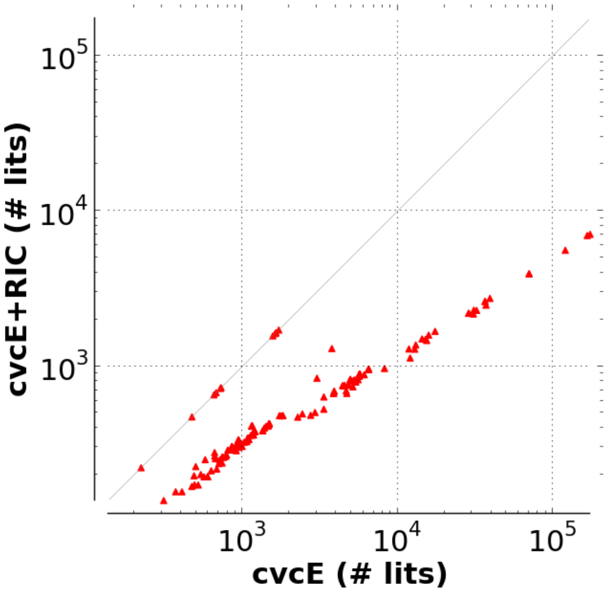

Figure 3.6: Literals in bit-blasted formula cvcE vs cvcE+RIC on mcm.

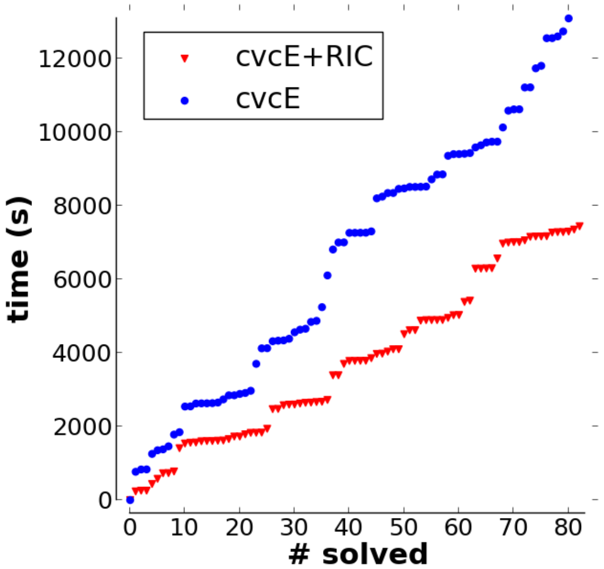

Figure 3.7: Run-time cvcE vs cvcE+RIC on mcm.

the problem in SAT.

Boolector is a specialized solver for bit-vectors and arrays and the winner of the 2014 SMT-COMP for the QF_BV and QF_ABV logics. For pure bit-vector formulas, it first employs word-level rewriting before bit-blasting the problem into AIG format, followed by conversion to CNF [21]. The rewrite rules it employs are split in three levels: simplification rewrites applied during formula construction, substitutions and static analysis rules and arithmetic normalization rules. Boolector also supports lazy instantiation of non-recursive first-order lambda functions (i.e. macros) [73]. Most other bit-vector solvers expand these eagerly. In order to reason about the combination of arrays and bit-vectors, Boolector incorporates efficient model driven lazy lemma instantiation in an abstraction refinement loop where array $read$ terms are abstracted to bit-vector variables. Recent work improves this approach by using the notion of relevancy to only refine the relevant part of the counter-example [70].

STP2 is another eager bit-vector solver specialized for handling the combination of arrays and bit-vectors. For bit-vector constraints, it first applies word-level simplifications and then encodes the problem to CNF via AIG [44]. It uses the abc [20] AIG package to apply AIG rewriting to the Boolean abstraction of the formula and then encodes the formula to CNF. One of the features that distinguishes STP2 from other bit-vector solvers, is its use of theory-level bit propagators [50]. For example, the following equality $a_{[8]} \& b_{[8]} = c_{[7]} \circ 1_{[1]}$ propagates that the least significant bit of a and b have to be 1: $b[0:0] = a[0:0] = 1_{[1]}$. This is a technique adapted from the CSP literature.

Although Z3 is a general purpose CDCL($\mathcal{T}$)-style SMT solver, its approach to deciding bit-vector formulas is most similar to that of eager solvers. Z3 first applies word-level rewrite techniques followed by eagerly encoding the formula into propositional logic via bit-blasting [32]. Z3 also uses a technique known as *relevancy* to reduce the size of the problem that is being reasoned about [31].

Yices [34] is another general purpose SMT solver with support for bit-vectors. Its bit-vector theory solver relies on rewriting to perform simplifications, then applies bit-blasting to all bit-vector operators with the exception of equality, for which it uses specialized reasoning. It handles arrays over bit-vectors via a model-based combination procedure.

The eager solver `cvcE` is similar to Yices and Z3 in that it is part of a general purpose SMT solver, but it differs in its use of a separate SAT solver for bit-vector constraints. Unlike the Z3 and Yices solvers, it uses a second SAT solver to reason exclusively about bit-vector constraints. From our experience, having a SAT solver exclusively dedicated to reasoning about bit-vector constraints had a significant performance impact.

Like STP2 `cvcE` relies on the `abc` AIG package to simplify the formula, but unlike STP2 we apply the simplifications to the entire formula and not just the Boolean abstraction. While this can be expensive for large formulas, we found it to be particularly beneficial for hardware equivalence problems. We found simplifying the entire formula helpful on equivalence checking problems. We are not aware of related work that implements a procedure similar to the word-level circuit refactoring described in Section 3.4.

Chapter 4

Lazy Bit-vector Solving

The standard technique for deciding the satisfiability of a $\mathcal{T}_{\mathsf{bv}}$ quantifier-free formula is to reduce the problem to Boolean satisfiability (SAT).This process is vividly called *bit-blasting*. Current state-of-the-art decision procedures for $\mathcal{T}_{\mathsf{bv}}$ build on bit-blasting by applying powerful word-level simplifications to the input formula before the final bit-blasting step. Chapter 3 described the implementation of one such eager bit-vector solver (cvcE) in the SMT solver CVC4. While often efficient in practice, this eager approach has several limitations: (i) the entire formula must be bit-blasted and solved at once, which may be difficult if the problem is too large; (ii) word-level structure and information can only be leveraged during preprocessing, not during solving; (iii) the complexity of the problem is a function of the bit-width; and (iv) eager solvers do not fit cleanly into theory combination frameworks. [1]

A *lazy* solver can address these limitations, explicitly targeting problems that are difficult for eager solvers and thus providing a complementary approach. In this chapter, we revisit the approach first proposed in [23,42] by extending and improving it in several

---

[1] While the "shell" solver approach mentioned in Section 3.1 provides a way of combining $\mathcal{T}_{\mathsf{bv}}$ with other theories, it does not easily lend itself to optimizing theory combination.

ways. The work in this chapter explores significant new ideas within the lazy framework, with the following contributions: (i) a dedicated SAT solver for $\mathcal{T}_{\mathsf{bv}}$ that supports bit-blasting-based propagation with lazy explanations; (ii) specialized $\mathcal{T}_{\mathsf{bv}}$ sub-solvers that reason about fragments of $\mathcal{T}_{\mathsf{bv}}$; (iii) inprocessing techniques to reduce the size of the bit-blasted formula when possible; and (iv) integration with the justification heuristics to minimize the number of literals sent to the bit-vector solver by the main SAT engine. The work in this chapter has been published in [48].

To evaluate the performance of the lazy approach, we implemented a lazy bit-vector solver, cvcLz, in the CVC4 SMT solver. Section 4.1 gives an overview of the architecture of cvcLz and how it is integrated in the CDCL($\mathcal{T}$) framework. Section 4.2 describes how to build an efficient incremental/backtrackable $\mathcal{T}_{\mathsf{bv}}$-solver using a second SAT solver SAT_{bb}. Maintaining the word-level structure enables the use of algebraic sub-solvers complete for certain fragments of $\mathcal{T}_{\mathsf{bv}}$. The implementation and performance impact of these algebraic sub-solvers is presented in Section 4.3. The word-level structure can additionally be exploited by applying word-level inprocessing during solving as described in Section 4.4). Section 4.5 describes some of the other techniques enabled by the lazy framework and their performance impact. Generating models in the lazy solver is briefly covered in Section 4.6. Finally we conclude by discussing related work in Section 4.7 .

4.1 Architecture

Designed for easy plug-and-play combination with solvers for other theories, the procedure integrates an online lazy $\mathcal{T}_{\mathsf{bv}}$ solver (LBV) into the CDCL($\mathcal{T}$) framework [71], separating theory-specific reasoning from the search over the Boolean structure of the input problem.

The lazy solver cvcLz combines algebraic, word-level reasoning with bit-blasting. Figure 4.1 shows the architecture of the lazy solver cvcLz. For simplicity, unless otherwise specified, we will assume the input formula ψ is a $\mathcal{T}_{\mathsf{bv}}$-formula, although the procedures described in this chapter also work on the combination of $\mathcal{T}_{\mathsf{bv}}$ with other theories. The input formula ψ first goes through the preprocessing phases described in Section 2.2. After CNF conversion, a CDCL-style SAT solver $\mathrm{SAT}_{\mathsf{main}}$ is used to search for a satisfying assignment to the Boolean abstraction of the formula. The SAT solver can query the justification heuristic $\mathrm{SAT}_{\mathcal{J}}$ described in Section 2.3 to stop early if the current partial assignment is satisfying. This helps reduce the size of the sub-problems the $\mathcal{T}_{\mathsf{bv}}$ solver has to reason about by considering only literals relevant in the current search context. Literals whose truth values are not required to satisfy the formula are ignored.

$\mathrm{SAT}_{\mathsf{main}}$ communicates this [partial] assignment to the theoryEngine, which in turn delegates $\mathcal{T}_{\mathsf{bv}}$ reasoning to the bit-vector theory. The bit-vector truth assignment A_{bv} is the sub-sequence of the trail A that contains only $\mathcal{T}_{\mathsf{bv}}$ literals (and so maintains the order in A). The set of $\mathcal{T}_{\mathsf{bv}}$ literals A_{bv} is asserted to the algebraic sub-solvers in order of efficiency and expressivity.

The lazy $\mathcal{T}_{\mathsf{bv}}$-solver is organized as a sequence of sub-solvers. If any of the sub-solvers identifies a conflict, the $\mathcal{T}_{\mathsf{bv}}$-solver can return *unsat* without querying the other sub-solvers. Similarly, if any of the sub-solvers is complete for the current truth assignment A_{bv}, the $\mathcal{T}_{\mathsf{bv}}$-solver can safely return *sat* without querying the other sub-solvers. While not a sub-solver, the In-processing module uses heuristic algebraic simplifications during search to detect word-level conflicts or reduce A_{bv} to *true*. If none of the algebraic engines detect a conflict or is sufficient to conclude satisfiability, the bit-blasting

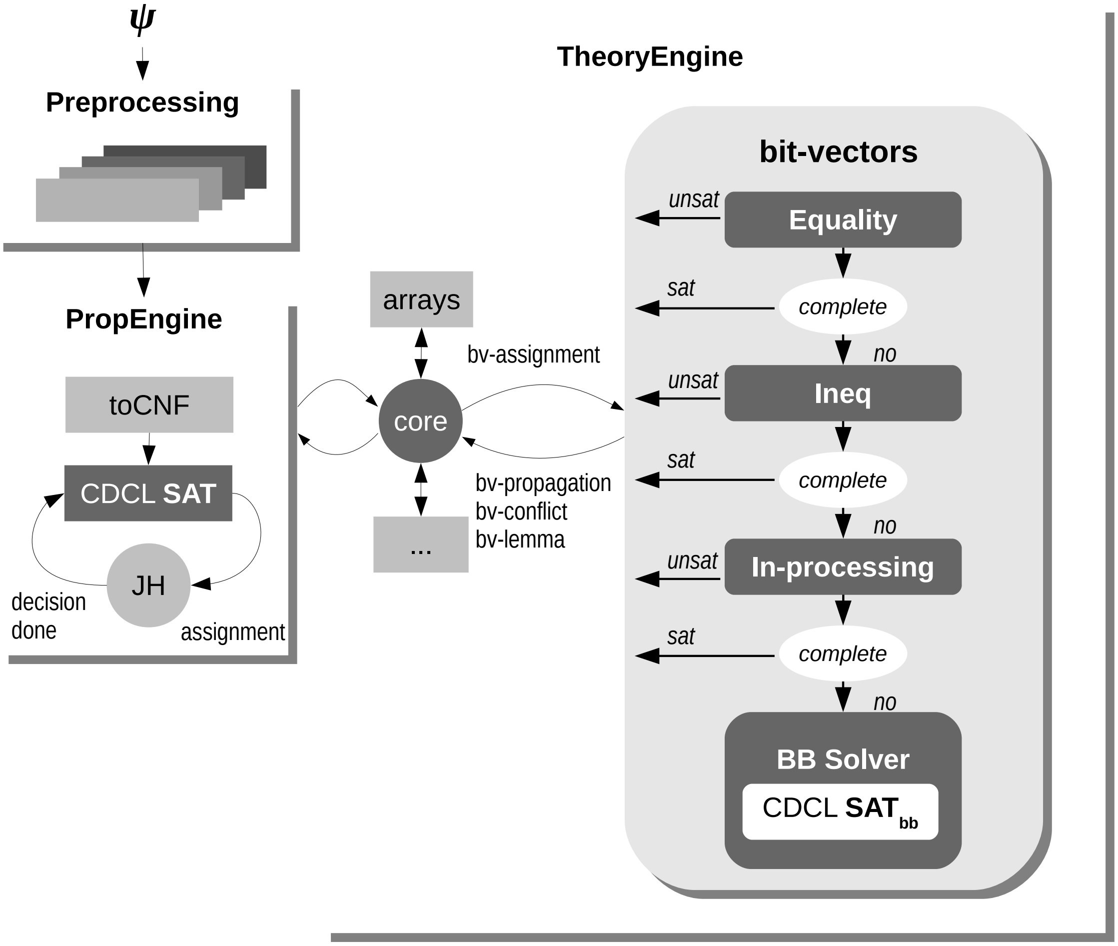

Figure 4.1: cvcLz architecture.

sub-solver $\mathsf{BV}_{\mathsf{bb}}$ provides a complete decision procedure for deciding A_{bv}.

4.2 Bit-blasting Solver

The bit-blasting solver $\mathsf{BV}_{\mathsf{bb}}$ is sufficient to decide the satisfiability of bit-vector constraints over the entire Σ_{bv} signature. At its heart is a second SAT solver SAT_{bb} distinct from the CDCL($\mathcal{T}$) Boolean engine $\text{SAT}_{\mathsf{main}}$. Our implementation is based on the open source MiniSAT 2.2.0 SAT solver [37]. We instrumented MiniSAT to efficiently implement the main requirements on a $\mathcal{T}$-solver: incrementality, conflict detection and propagation of entailed literals.

Incrementality Because A_{bv} is a sub-sequence of the trail A, literals are pushed and popped in an incremental fashion. It is important for $\mathsf{BV}_{\mathsf{bb}}$ to efficiently reason incrementally in this manner. Most SAT solvers do not have full support for incremental solving. While CDCL-style SAT solvers use backtracking to pop their trail, the set of input clauses is usually fixed. Most solvers easily support adding more input clauses during solving, a feature we take advantage of. However, efficiently removing problem clauses is an area of active research, as removed clauses may have participated in the derivation of learned clauses [39,68]. Bit-blasting almost any $\mathcal{T}_{\mathsf{bv}}$-atom requires multiple clauses, and backtracking $\mathsf{BV}_{\mathsf{bb}}$ would require removing input clauses. Incrementality can be simulated in a CDCL SAT solver using the *solve with assumptions* [37] feature described in Section 1.4.3. Given a fixed set of input clauses C, the SAT solver can check its satisfiability assuming a given set of literals are *true* (the assumptions).

We exploit this feature by associating to each $\mathcal{T}_{\mathsf{bv}}$-atom a in the input formula ψ a corresponding *marker variable* a^{BB} such that a holds iff a^{BB} is assigned to true. We denote by $\mathsf{BB}(\psi) = \{a^{\mathsf{BB}}, \neg a^{\mathsf{BB}} \mid a \text{ is an atom in } \psi\}$ the set of all marker literals in a $\mathcal{T}_{\mathsf{bv}}$ formula ψ. Let bbAtom be the bit-blasting function that takes a $\mathcal{T}_{\mathsf{bv}}$-atom a and returns an equisatisfiable Boolean formula. (See Section 3.2 for a description of how bbAtom works.) Before starting search, we assert the following formula to SAT_{bb}:

$$\bigwedge_{\text{atom } a \in \psi} \left(a^{\mathsf{BB}} \Leftrightarrow \mathsf{bbAtom}(a)\right) .$$

We will denote by C^{BB} the bit-blasting clauses corresponding to the above formula. Since these are *definitional* clauses they are satisfiable by construction: because each

a_{BB} is unconstrained the logical equivalences can always be satisfied.

Given bit-vector constraints A_{bv} asserted to $\mathsf{BV}_{\mathsf{bb}}$, the corresponding call to $\mathrm{SAT}_{\mathsf{bb}}$ takes as assumptions $\mathsf{A}_{\mathsf{bv}}^{\mathsf{BB}} = \{l^{\mathsf{BB}} | l \in \mathsf{A}_{\mathsf{bv}}\}$ where l^{BB} is the marker literal corresponding to literal l.

When $\mathrm{SAT}_{\mathsf{main}}$ backtracks during conflict resolution, it ensures that $\mathrm{SAT}_{\mathsf{bb}}$ also backtracks by calling the backtrack procedure in Algorithm 4 in Section 1.4.3. Note that the decision levels in the two solvers do not match: one level in $\mathrm{SAT}_{\mathsf{main}}$ may correspond to multiple levels in $\mathrm{SAT}_{\mathsf{bb}}$ and vice-versa. For example one round of BCP in $\mathrm{SAT}_{\mathsf{main}}$ may propagate multiple $\mathcal{T}_{\mathsf{bv}}$-literals at the same decision level. When asserted in $\mathrm{SAT}_{\mathsf{bb}}$, each will correspond to its own decision level. On the other hand, if more than one theory is involved, it may be the case that no $\mathcal{T}_{\mathsf{bv}}$-literals are asserted at some decision level d in $\mathrm{SAT}_{\mathsf{main}}$. In this case the decision level in $\mathrm{SAT}_{\mathsf{bb}}$ would stay the same while that in $\mathrm{SAT}_{\mathsf{main}}$ would increase.

Conflict generation If A_{bv} is unsatisfiable, $\mathsf{C}^{\mathsf{BB}} \wedge \mathsf{A}_{\mathsf{bv}}^{\mathsf{BB}}$ must also be unsatisfiable. $\mathrm{SAT}_{\mathsf{bb}}$ can infer a (non-minimal) inconsistent subset of A_{bv} via resolution. As C^{BB} is always satisfiable, the only way $\mathsf{A}_{\mathsf{bv}}^{\mathsf{BB}} \wedge \mathsf{C}^{\mathsf{BB}}$ can be unsatisfiable is if one of the assumption literals $l^{\mathsf{BB}} \in \mathsf{A}_{\mathsf{bv}}^{\mathsf{BB}}$ is falsified by a subset of the other assumptions.[2] The assumption literals corresponding to the conflict can be computed by resolving backwards from the falsified assumption, until all the literals in the conflict clause are currently assigned assumption literals. Algorithm 9 takes as its input the falsified literal p and computes the conflict clause learned. All literals $l \in$ learned are over variables in $\mathsf{A}_{\mathsf{bv}}^{\mathsf{BB}}$ and the corresponding $\mathcal{T}_{\mathsf{bv}}$ clause is $\mathcal{T}_{\mathsf{bv}}$-valid. The procedure is similar to analyzeConflict in Algorithm 3. Because we did not make any "real" decisions, all literals that do not have

---

[2] The subset may be empty if a literal itself is contradictory e.g. $x_{[8]} < 0_{[8]}$.

a reason must either be assumptions or units. The assumptions remain in learned, while the units are resolved out. If assumption a_1 entails another a_2 via BCP, the a_1 assumption will be included in the conflict i.e. we resolve down to the earliest assumption.

Algorithm 9: Assumptions conflict.

```
Input: $p$ falsified assumption literal
learned $\leftarrow$ reason$(p)$;
seen $\leftarrow \emptyset$ ;
while learned $\setminus$ seen $\neq \emptyset$ do
    $lit \leftarrow \arg\max_l \{index_{\mathsf{A}}(l) | l \in$ learned $\setminus$ seen$\}$;
    if reason$(lit) \neq \bot$ then
        learned $\leftarrow$ Res(learned, reason$(lit)$);
    else if $lit \notin$ BB$(\psi)$ then
        learned $\leftarrow$ Res(learned, $\{\neg lit\}$);
    seen $\leftarrow$ seen $\cup \{lit\}$;
return learned ;
```

Propagation To infer that a $\mathcal{T}_{\mathsf{bv}}$-literal l is $\mathcal{T}_{\mathsf{bv}}$-entailed by A_{bv}, the SAT_{bb} solver uses only Boolean-constraint propagation (BCP) and makes no "real" decisions. We instrumented the SAT_{bb} solver to give explanations using its conflict resolution infrastructure resolving backwards from the propagated literal l in a way similar to Algorithm 9. Algorithm 11 shows the interaction between $\mathsf{BV}_{\mathsf{bb}}$ and SAT_{bb}. Because a complete satisfiability check in SAT_{bb} can be very expensive, we only call Solve in final calls to check. If $\mathcal{T}_{\mathsf{bv}}$-check$_{bb}$ is not final, we rely on efficient bit-level reasoning to detect easy conflicts and propagations using BCP.

For efficiency, it is important that a theory be able to explain the propagations in a lazy fashion. However, requesting an explanation after search has already been done in SAT_{bb} can lead to SAT_{bb} being in a state in which the explanation cannot be computed anymore. Example 14 illustrates how this can happen. The example heavily relies on concepts introduced in Section 1.4.

Example 14. We will show how the explanation of an assumption literal can be lost after SAT$_{\mathsf{bb}}$ backtracks during subsequent search. This problem is not specific to clauses coming from bit-blasting $\mathcal{T}_{\mathsf{bv}}$-atoms, but applies to solve-with-assumptions CDCL SAT solvers in general.

Assume we are starting with the following bit-blasted clauses C^{BB}:

$$\begin{array}{rccc}
c_1: & u & p_2 & \\
c_2: & u & p_3 & \\
c_3: & \neg p_2 & \neg p_3 & \neg a_2 \\
c_4: & \neg a_1 & p_0 & \\
c_5: & \neg p_0 & \neg u & \neg p_1 \\
c_6: & \neg u & p_1 & \\
c_7: & \neg d & u & \\
c_8: & \neg a_1 & \neg a_2 & a_3
\end{array}$$

The a_1, a_2 and a_3 literals are marker literals: $a_1, a_2, a_3 \in \mathsf{BB}(\psi)$. During the first conflict the d literal will correspond to the "real" decision, the u literal will be the UIP and the p_1, p_2, p_3 literals will be propagated literals. Note that these roles only hold for the first conflict and will change as the solver backtracks.

The following table shows the trail of SAT$_{\mathsf{bb}}$ after a non-final call to the check procedure of $\mathsf{BV}_{\mathsf{bb}}$, $\mathcal{T}_{\mathsf{bv}}$-$\mathsf{check}_{bb}([a_1, a_2], \mathit{false})$:

Level	1		2	
Trail	a_1	p_0	a_2	a_3
Reason	$\bot$	c_4	$\bot$	c_8

Because a_3 is a marker literal that is entailed by the current assumptions, we can propagate it out to SAT$_{\mathsf{main}}$. Note that at this point we could compute its explanation using the reason map as $explain(a_3) : a_1 \wedge a_2$. Say SAT$_{\mathsf{main}}$ does not find any conflicts, and the next call to $\mathcal{T}_{\mathsf{bv}}$-$\mathsf{check}_{bb}$ is final: $\mathcal{T}_{\mathsf{bv}}$-$\mathsf{check}_{bb}([a_1, a_2], \mathit{true})$. Since there are no more assumptions to process, SAT$_{\mathsf{bb}}$ can now make "real" decisions. Say it decides literal d. This leads to a conflict as shown by the following trail:

Level	1		2		3			
Trail	a_1	p_0	a_2	a_3	d^d	u	p_1	$\neg p_1$
Reason	$\bot$	c_4	$\bot$	c_8	$\bot$	c_7	c_6	c_5

The corresponding implication graph is shown in Figure 4.2. Conflict resolution proceeds as usual and identifies the first UIP u, leading to learning the following clause:

$$c_{10} : \quad \neg u \quad \neg p_0$$

with the corresponding backtracking level of 1. The trail below shows the SAT$_{\mathsf{bb}}$ state after backtracking to level 1 and a round of BCP (see Figure 4.3 for the corresponding implication graph):

Level	1							2
Trail	a_1	p_0	$\neg u$	$\neg d$	p_2	p_3	$\neg a_2$	a_2
Reason	$\bot$	c_4	c_{10}	c_7	c_1	c_2	c_3	$\bot$

Because the new learned clause increased the propagation power of the clause database, we discover a conflict before a_3 is propagated again. Returning this conflict to SAT$_{\mathsf{main}}$ will force SAT$_{\mathsf{main}}$ to do its own conflict resolution, which may require an explanation for the propagated literal a_3. However, in the current state of SAT$_{\mathsf{bb}}$, a_3 is unassigned. The problem is that during search, the learned clauses strengthen the

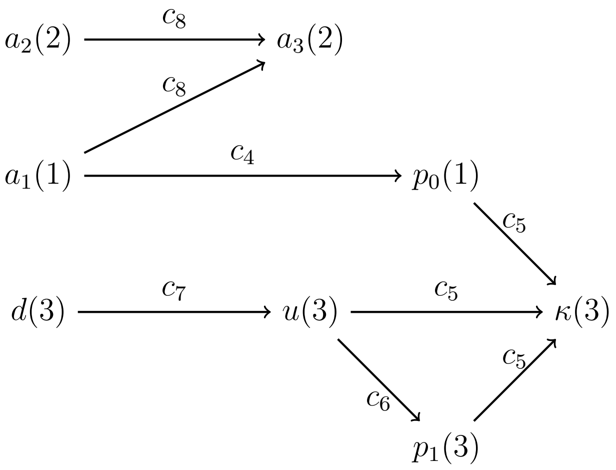

Figure 4.2: Implication graph for Example 14 (first conflict).

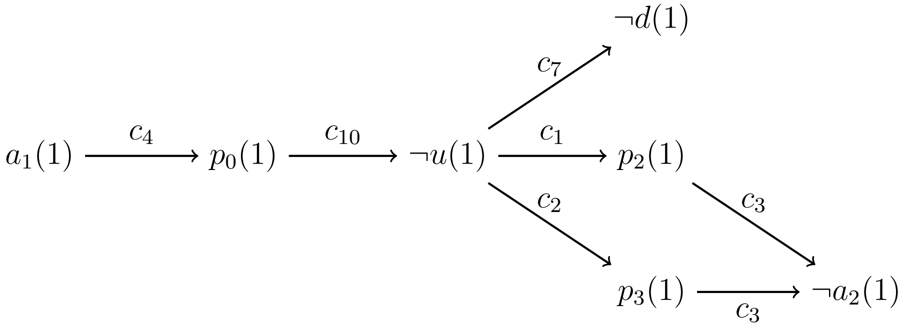

Figure 4.3: Implication graph for Example 14 (second conflict).

propagation power of the clause data-base. There is no guarantee that the order in which assumption literals are propagated is maintained.

To ensure that we do not "lose" the explanation of any propagated literal, we need to be careful about backtracking in SAT$_{\mathsf{bb}}$ when resolving a conflict. We will define the *assumption level* $\mathsf{assumpLevel}$ as the highest decision level that corresponds to currently assigned assumptions $\mathsf{A}^{\mathsf{BB}}_{\mathsf{bv}}$. If all the assumptions have been processed, this is equal to the number of assumptions assump. Algorithm 10 shows the changes that need to be made to the CDCL with assumptions (Algorithm 4) presented in Section 1.4.3 in order to enable lazy explanations. When backtracking below $\mathsf{assumpLevel}$ due to a learned clause L, we now check whether the clause data-base strengthened by L would now lead to a conflict. This is done by speculatively running BCP. If BCP does not identify a conflict, it is safe to backtrack to level b: all the $\mathcal{T}_{\mathsf{bv}}$ propagations will be propagated again and can be explained. If BCP finds a conflict, it means that adding L proves the assumptions are unsatisfiable. Recall that the learned clause L is an asserting clause: it forces the UIP to be asserted in a polarity that is the opposite of what it was before. If BCP finds a conflict, it means that both polarities of the UIP led to a conflict given the current assumptions, so the problem must be unsatisfiable.

Experimental evaluation. Experiments in this chapter were ran on the StarExec [80] cluster infrastructure with a timeout of 900 seconds and a memory limit of 50GB.[3] The benchmark set consists of all 32509 QF_BV problems in SMT-LIB 2014-06-03. All scatter plots are on a log-scale, and the x and y-axis represent CPU time in seconds, unless otherwise specified. We will denote by cvcLz the best configuration of the lazy solver LBV implemented in CVC4, with all features described in Section 4.1 enabled. To denote that a feature is disabled, we will use the $-$ and to indicate that the feature is enabled we will use $+$. For example cvcLz-P denotes cvcLz with propagation disabled, while cvcLz+EP denotes cvcLz with eager computation of propagation explanations.

Figure 4.4 shows the impact of $\mathsf{BV}_{\mathsf{bb}}$ propagation with lazy explanations. This feature is essential for performance and leads to more problems solved in less time. To explore whether computing explanations lazily for $\mathsf{BV}_{\mathsf{bb}}$ makes a difference in performance, we implemented a version that computes the explanation eagerly (cvcLz+EP). As Figure 4.5 illustrates, this leads to a significant degradation in performance. Note that the explanation computed eagerly may differ from the explanation computed lazily.

---

[3] Experiments were ran on the queue all.q consisting of Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz machines with 268 GB of memory.

Algorithm 10: CDCL with assumptions and lazy explanations.

```
Input: $\langle \psi, \text{assump} \rangle$ input formula and assumptions
C $\leftarrow$ toCnf($\psi$);
$\langle$A, dl$\rangle \leftarrow \langle [], 0 \rangle$ ;
while true do
    c $\leftarrow$ BCP(C, A);
    if c $\neq$ undef then
        $\langle b, L \rangle \leftarrow$ analyzeConflict(c) ;
        C $\leftarrow$ C $\cup L$;
        if $b = -1$ then
            return unsat;
        if $b <$ assumpLevel then
            backtrack(assumpLevel, A);
            c $\leftarrow$ BCP(C, A);
            if c $\neq$ undef then
                return unsat;
        backtrack($b$, A);
        continue;
    if allAssigned(A) then
        return sat;
    if hasAssumps(A, assump) then
        $l \leftarrow$ nextAssump(A, assump) ;
        if A$(l) = false$ then
            return unsat;
        if A$(l) = true$ then
            dl $\leftarrow$ dl $+ 1$;
    else
        $l \leftarrow$ decide();
    dl $\leftarrow$ dl $+ 1$;
    A $\leftarrow$ A :: $l$;
```

```
Algorithm 11: $\mathcal{T}_{\mathsf{BV}}$-check$_{bb}$
Input: $\langle \mathsf{A}, \mathsf{final} \rangle$
$\langle \mathsf{P}, \mathsf{L} \rangle \leftarrow \mathsf{BCP}(\mathsf{A})$ ;
if final and $\mathsf{L} = \emptyset$ then
    $\mathsf{L} \leftarrow \mathsf{Solve}(\mathsf{A})$ ;
return $\langle \mathsf{P}, \mathsf{L} \rangle$ ;
```

This is due to the fact that the clauses learned in-between assigning the propagated literal and requesting the explanations can change the propagation order in SAT_{bb}. This explains the presence of the points above the diagonal for which the different explanation changed the search. We examined the ratio of $\mathsf{BV}_{\mathsf{bb}}$ propagated literals that require an explanation to the number of $\mathsf{BV}_{\mathsf{bb}}$ propagated literals. Out of the 32590 QF_BV benchmarks, only 83 required explaining more than 1% of the propagated literals, with the highest percent of explained propagation being 63%. Only 1719 problems required any explanations from $\mathsf{BV}_{\mathsf{bb}}$.[4]

4.3 Algebraic Sub-solvers

The LBV solver consists of four sub-solvers, each sufficient to decide the satisfiability of constraints in the sub-solver's own fragment of $\mathcal{T}_{\mathsf{bv}}$ (see Table 4.1): the equality solver $\mathsf{BV}_{\mathsf{eq}}$, the core solver $\mathsf{BV}_{\mathsf{core}}$, the inequality solver $\mathsf{BV}_{\mathsf{ineq}}$, and the bit-blasting solver $\mathsf{BV}_{\mathsf{bb}}$. (See Table 1.4 in Section 1.3 for the precise definitions of Σ_{eq}, Σ_{con}, Σ_{ineq}. The Σ_{bv} signature is just the full $\mathcal{T}_{\mathsf{bv}}$ signature.) Each sub-solver is incremental and provides the theory solver functionalities described in Section 1.5. Our current implementation uses $\mathsf{BV}_{\mathsf{eq}}$ by default, but it can alternatively use $\mathsf{BV}_{\mathsf{core}}$. The architecture of LBV was designed to be modular and extensible: all the bit-vector reasoning is confined within

---

[4] Part of the reason is that the benchmark selection is dominated by the sage family that contains 26K problems out of the 32K in QF_BV. Most of these problems are easy.

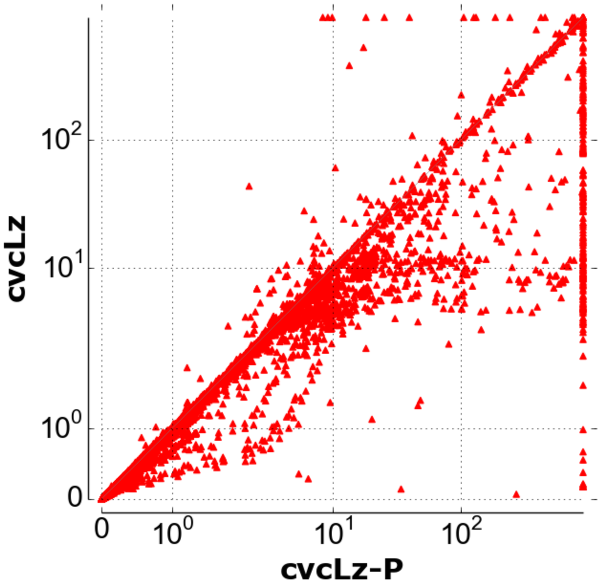

Figure 4.4: Disabling propagation (cvcLz-P).

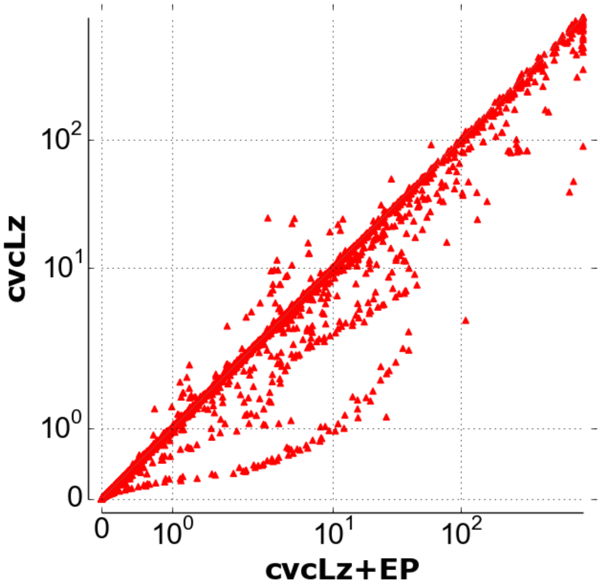

Figure 4.5: Computing eager explanations (cvcLz+EP).

the solver, and it is easy to enhance it by adding more sub-solvers.

Algorithm 12: $\mathcal{T}_{\mathsf{BV}}$-check

```
Input: $\langle \mathsf{A}, \mathsf{final} \rangle$
$\langle \mathsf{P_{eq}}, \mathsf{L_{eq}}, \mathsf{done} \rangle \leftarrow \mathcal{T}_{\mathsf{bv}}\text{-}\mathsf{check}_{eq}(\mathsf{A}, \mathsf{final})$ ;
if done then
    return $\langle \mathsf{P_{eq}}, \mathsf{L_{eq}} \rangle$ ;
$\langle \mathsf{P_{ineq}}, \mathsf{L_{ineq}}, \mathsf{done} \rangle \leftarrow \mathcal{T}_{\mathsf{bv}}\text{-}\mathsf{check}_{ineq}(\mathsf{A}; \mathsf{P_{eq}}, \mathsf{final})$ ;
if done then
    return $\langle \mathsf{P_{eq}}; \mathsf{P_{ineq}}, \mathsf{L_{ineq}} \rangle$ ;
$\langle \mathsf{P_{bb}}, \mathsf{L_{bb}}, \mathsf{done} \rangle \leftarrow \mathcal{T}_{\mathsf{bv}}\text{-}\mathsf{check}_{bb}(\mathsf{A}; \mathsf{P_{eq}}; \mathsf{P_{ineq}}, \mathsf{final})$ ;
return $\langle \mathsf{P_{eq}}; \mathsf{P_{ineq}}; \mathsf{P_{bb}}, \ \mathsf{L_{eq}} \cup \mathsf{L_{ineq}} \cup \mathsf{L_{bb}} \rangle$
```

Algorithm 12 shows the implementation of $\mathcal{T}_{\mathsf{bv}}$-check, the $\mathcal{T}$-check from Algorithm 5 corresponding to the LBV solver. Given a partial assignment $\mathsf{A_{bv}}$, $\mathcal{T}_{\mathsf{bv}}$-check returns a set of theory literals entailed by $\mathsf{A_{bv}}$, and a set of $\mathcal{T}$-valid clauses. $\mathcal{T}_{\mathsf{bv}}$-check calls the subsolvers in increasing order of computational cost. For each $i \in \{\mathsf{eq}, \mathsf{ineq}, \mathsf{bb}\}$, $\mathcal{T}_{\mathsf{bv}}$-check$_i$ returns a sequence P_i of propagated literals, a set L_i of learned clauses, and a Boolean value indicating whether the solver is done or not. A solver i is done if $\mathcal{T}_{\mathsf{bv}}$-check$_i$ has detected an inconsistency, or if it can determine that $\mathsf{A_{bv}}$ is consistent, i.e. all literals of $\mathsf{A_{bv}}$ fall in the sub-solver's fragment of $\mathcal{T}_{\mathsf{bv}}$. If no solver detects an inconsistency, $\mathcal{T}_{\mathsf{bv}}$-check returns the collection of all the propagated literals and lemmas generated by the individual sub-solvers.

The sub-solvers process all literals in $\mathsf{A_{bv}}$. However, except for $\mathsf{BV_{bb}}$, they reason on an abstraction of the literals. In particular, $\mathsf{BV_{eq}}$ treats all function and predicate symbols other than $=$ as uninterpreted, while $\mathsf{BV_{ineq}}$ (as well as $\mathsf{BV_{core}}$) treats as fresh

Table 4.1: LBV sub-solver signatures

$\mathsf{BV_{eq}}$	$\mathsf{BV_{core}}$	$\mathsf{BV_{ineq}}$	$\mathsf{BV_{bb}}$
Σ_{eq}	$\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{con}}$	$\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{ineq}}$	Σ_{bv}

variables any function or predicate symbols of arity greater than one which are not in its signature.

4.3.1 Equality Solver

The equality solver $\mathsf{BV}_{\mathsf{eq}}$, corresponding to $\mathcal{T}_{\mathsf{bv}}$-$\mathsf{check}_{eq}$, uses a variant of well-known incremental polynomial-time congruence-closure (CC) algorithms [33] to decide the satisfiability of its constraints. Standard CC algorithms assume that sorts have an unbounded cardinality. This makes them incomplete for reasoning about equality and disequality constraints in $\mathcal{T}_{\mathsf{bv}}$. For example, the formula $x_{[1]} \neq y_{[1]} \wedge x_{[1]} \neq z_{[1]} \wedge y_{[1]} \neq z_{[1]}$ is not satisfiable in $\mathcal{T}_{\mathsf{bv}}$ because there are only two distinct bit-vectors of width 1.

The problem of deciding conjunctions of disequalities between terms of sort $[n]$ in $\mathcal{T}_{\mathsf{bv}}$ is NP-complete. It is equivalent to graph coloring: nodes can be encoded using bit-vectors and edges as disequalities over bit-vectors. There is a satisfying assignment if the nodes in the graph can be colored with 2^n colors such that no two adjacent nodes share the same color.

We handle the finite cardinality of the bit-vector sorts by trying to build a satisfying valuation for all the terms in a given Σ_{eq}-constraint. In final calls to check, once the CC algorithm is done and has not detected any inconsistency, $\mathsf{BV}_{\mathsf{eq}}$ attempts to assign a distinct constant value to each congruence class $c^0_{[n]}, \ldots, c^k_{[n]}$ for each sort $[n]$ in the input problem. If this is not possible, which means that $k > 2^n$, it returns a lemma of the form

$$\bigvee_{0 \leq i < j \leq 2^n} r^i_{[n]} = r^j_{[n]}$$

where r_i is a representative for class $c^i_{[n]}$, stating that at least two of the first $2^n + 1$ congruence classes must be merged.

This process continues until, either the splits lead to an inconsistency or the sub-solver finds a satisfying valuation. The cardinality lemmas are currently generated only if the congruence classes consist just of bit-vector constants and variables. While just guessing a merge of congruence classes is a fairly unsophisticated way to deal with sort cardinality constraints, we found that it works well in practice for bit-vectors. The reason is simply that the cardinality of a sort $[n]$ grows exponentially with n and so for large enough bit-widths one needs to have a very large number of disequalities in the input problem to force a merge.

4.3.2 Inequality Solver

The inequality solver $\mathsf{BV}_{\mathsf{ineq}}$ can decide the satisfiability of $(\Sigma_{\mathsf{eq}} \cup \Sigma_{\mathsf{ineq}})$-constraints. It relies on an incremental special-purpose algorithm for deciding the satisfiability of conjunctions of bit-vector inequalities.[5] $\mathsf{BV}_{\mathsf{ineq}}$ only needs to reason about $<$ and $\leq$ since equality can be expressed in terms of $\leq$, and inequalities of the form $x_{[n]} \neq y_{[n]}$ can be eliminated by requesting the following split:

$$x_{[n]} = y_{[n]} \vee x_{[n]} < y_{[n]} \vee y_{[n]} < x_{[n]}.$$

These lemmas are generated by $\mathcal{T}_{\mathsf{bv}}\text{-}\mathsf{check}_{ineq}$ in final checks, only if A is a constraint over $\Sigma_{eq} \cup \Sigma_{ineq}$.

For the rest of this section we assume all inequalities are unsigned. A similar approach can be used for signed inequalities. We will use $\lhd$ to denote both $<$ and $\leq$, and $\lhd^*$ for the transitive closure of $\lhd$. In the following, we will use $\mathcal{I}$ to denote a conjunction of inequality constraints over variables and constants of the same sort $[n]$.

---

[5] This problem is a special case of modular difference logic that can be reduced to integer difference logic, as there is no wrap-around behavior due to overflows.

Definition 8. Given the set of inequalities $\mathcal{I}$, an *$\mathcal{I}$-valuation* M is a mapping from bit-vector variables $v_{[n]} \in \mathcal{I}$ to constant values $c_{[n]}$.

For convenience we extend M to map constants to themselves, and other bit-vector terms and formulas as expected.[6] A valuation M *satisfies* a bit-vector constraint ϕ if $\mathsf{M}(\phi) = true$.

Definition 9. We define a partial order $\preceq$ over valuations on $\mathcal{I}$ as follows: given valuations M_1 and M_2, $\mathsf{M}_1 \preceq \mathsf{M}_2$ iff for all bit-vector terms $t \in I$, $\mathsf{M}_1(t) \leq \mathsf{M}_2(t)$.

We say a valuation M is the *least valuation* of $\mathcal{I}$ if M satisfies $\mathcal{I}$ and for all valuations M' satisfying $\mathcal{I}$, $\mathsf{M} \preceq \mathsf{M}'$.

Lemma 1. Any satisfiable set of inequalities $\mathcal{I}$ has a least valuation.

Proof. We show that $\preceq$ has a least element over the set of all satisfying valuations of $\mathcal{I}$. Given two satisfying $\mathcal{I}$ valuations, M_1 and M_2, we define the $\sqcap$ operator as follows:[7]

$$(\mathsf{M}_1 \sqcap \mathsf{M}_2)(t) := \min(\mathsf{M}_1(t), \mathsf{M}_2(t)).$$

Next we show that the new valuation $\mathsf{M} = \mathsf{M}_1 \sqcap \mathsf{M}_2$ satisfies $\mathcal{I}$. Let $x \lhd y$ be an inequality in $\mathcal{I}$. We show that $\mathsf{M}(x) \lhd \mathsf{M}(y)$:

($\lhd$ is $<$) : Because M_1 and M_2 are satisfying, $\mathsf{M}_1(x) < \mathsf{M}_1(y)$ and $\mathsf{M}_2(x) < \mathsf{M}_2(y)$. As $\mathsf{M}(x) = \min(\mathsf{M}_1(x), \mathsf{M}_2(x))$ we have $\mathsf{M}(x) \leq \mathsf{M}_1(x)$ and $\mathsf{M}(x) \leq \mathsf{M}_2(x)$. If $\mathsf{M}_1(y) \leq \mathsf{M}_2(y)$ we have $\mathsf{M}(y) = \mathsf{M}_1(y)$ which entails $\mathsf{M}(x) < \mathsf{M}(y)$ (the inequality is satisfied). Similarly for the $\mathsf{M}_1(y) > \mathsf{M}_2(y)$ case.

($\lhd$ is $\leq$) : Analogous to previous case.

---

[6] In our case M will only apply to variables, constants and inequalities.

[7] It can be shown that $\langle \preceq, \sqcap, \sqcup \rangle$ form a lattice over all the satisfying valuations of inequality set $\mathcal{I}$, where $\sqcup$ is defined using $\max$

By construction $\mathsf{M} \preceq \mathsf{M}_1$ and $\mathsf{M} \preceq \mathsf{M}_2$. Let S be the set of satisfying valuations of $\mathcal{I}$. Since bit-vectors are bounded, S is finite. Let the cardinality of $|S|$ be N. Let $\mathsf{M}_1, \ldots, \mathsf{M}_N$ be an enumeration over S, and M'_i be the result of applying $\sqcap$ to the first i valuations:

$$\mathsf{M}'_i = ((\ldots(\mathsf{M}_1 \sqcap \mathsf{M}_2) \sqcap \mathsf{M}_3) \ldots \sqcap \mathsf{M}_i)$$

Inductively, each M'_i is a least valuation: each $\mathsf{M}'_{i+1} \preceq \mathsf{M}'_i$. Therefore the least valuation of $\mathcal{I}$ is M'_N. □

Algorithm 13: ProcessInequality

```
Input: $a \lhd b$
$\mathcal{I} \leftarrow \mathcal{I} \cup \{a \lhd b\}$ ;
val ← M;
O ← M;
val(b) ← newVal(a ◁b);
if not val(b) = M (b) then
    PQ.push(b) ;
while not PQ.empty() do
    x ← PQ.pop() ;
    fail ← updateModel(x, val(x)) ;
    if fail then
        return ⟨ unsat ,buildConflict(x)⟩ ;
    foreach x ◁ y ∈ $\mathcal{I}$ do
        val(y) ← max(val(y), newVal(x ◁y));
        if not val(y) = M (y) then
            PQ.push(y) ;
return ⟨ sat ,∅⟩ ;
```

The starting model M is defined as:

$$\mathsf{M}(v_{[n]}) = 0_{[n]}.$$

Algorithm 14: updateModel

```
Input: $\langle x, newX \rangle$
/* Cannot update a constant */
if x is a constant and $x \neq newX$ then
    return true;
/* Overflow */
if $newX < x$ then
    return true;
/* Cycle containing a non-strict inequality */
if x = a then
    return true;
M (x) ← newX;
return false;
```

where $0_{[n]}$ is the binary representation of 0 in n bits. We maintain the invariant that M is the least model of $\mathcal{I}$. Given a new inequality $a \lhd b$, we want to extend M to a least model of $\mathcal{I} \cup \{a \lhd b\}$, or discover that the problem is unsatisfiable. If $\mathsf{M}(a) \lhd \mathsf{M}(b)$ already holds, we are done. Otherwise, the least model property guarantees that terms a and b have the least possible values. Therefore in order to satisfy $a \lhd b$ we must increase b's value, if possible, to match that of a. The update cannot violate previously satisfied inequalities of the form $\{t_1 \lhd t_2 \mid t_2 \lhd^* b\}$. The only terms whose values may need to further be updated are terms t such that $b \lhd^* t$.

Given a set of inequalities $\mathcal{I}$, $\mathsf{BV}_{\mathsf{ineq}}$ builds the least model incrementally by processing the inequalities using the ProcessInequality procedure described in Algorithm 13. The procedure stores its inequalities as a set in the variable $\mathcal{I}$, initialized to $\emptyset$. The valuation val maps each x to an intermediate value $\mathsf{val}(x)$ that x should have in order to satisfy $\mathcal{I}$: $\mathsf{val}(x)$ can be seen as refining the lower bound on the final value of x. Initially val is equal to M. The original M is also saved in the O variable.

Assuming a is assigned $\mathsf{M}(a)$, the $\mathsf{newVal}(a \lhd b)$ procedure returns the smallest value b can have in order to satisfy $a \lhd b$. If $\mathsf{M}(a) \lhd \mathsf{M}(b)$, then newVal just returns the current $\mathsf{M}(b)$. Otherwise, if $\lhd$ was $\leq$ then newVal will return $\mathsf{M}(a)$, and if it was $<$, it will return $\mathsf{M}(a) + 1$.[8]

If the value required to satisfy $a \lhd b$ is different from the current valuation $\mathsf{M}(b)$, b is pushed onto the priority queue PQ. The priority queue PQ reduces the number of times the value of each term is updated, by prioritizing terms with lower original model value: it returns the element x with the least valuation $\mathsf{O}(x)$. If there are two elements x and y that have the same valuation $\mathsf{O}(x) = \mathsf{O}(y)$, PQ returns the element with the highest val. Note that the valuations M and val change through the while-loop iterations, while O stays the same.

During each iteration of the loop, an element x is popped from the queue. The updateModel procedure shown in Algorithm 14 attempts to update the valuation of $\mathsf{M}(x)$ to $\mathsf{val}(x)$. There are several ways in which this can fail: x is a constant and $x \neq \mathsf{val}(x)$ (recall that newVal returns the original value if the inequality is satisfied); increasing the value of x leads to an overflow (in other words $x > \mathsf{val}(x)$); or there is a cyclic sequence of inequalities with at least one strict inequality in it. Since ProcessInequality is called when M is satisfying, we know that no such cycles could have already existed in the graph. Therefore the $a \lhd b$ edge must have formed the cycle. The updateModel procedure detects such cycles by checking whether $x = a$.

If updateModel fails, *unsat* is returned along with a conflict built by traversing the graph backwards and collecting the reason for each value update. If updateModel succeeds, we examine all the successors y of x, and compute the least value they need to be updated to, to satisfy the $x \lhd y$ edges. Note that y may already have required updating due to some other incoming edge: we take the $\max$ of the two update values. If the

---

[8] Overflows will be caught by updateModel when it checks that $\mathsf{M}(x) \leq \mathsf{val}(x)$.

valuation of y needs to be updated, we push y on the priority queue. The priority queue PQ has the following invariant: all the nodes $x \in \mathsf{PQ}$ must be updated to at least $\mathsf{val}(x)$ in order to satisfy $\mathcal{I}$.

Claim 2. Given a least model M for a set of inequalities $\mathcal{I}$, the least model M' for any other set of inequalities $\mathcal{I}' \supseteq \mathcal{I}$ must satisfy $\mathsf{M} \preceq \mathsf{M}'$.

Proof. Any M' that satisfies $\mathcal{I}'$ also satisfies $\mathcal{I}$. As M is the least model of $\mathcal{I}$, $\mathsf{M} \preceq \mathsf{M}'$. □

Claim 3. During the execution of Algorithm 13, val and M maintain the invariant that they are less than the least valuation L of $\mathcal{I}$: $\mathsf{val} \preceq \mathsf{L}$ and $\mathsf{M} \preceq \mathsf{L}$.

Proof. We will prove this by induction on the number i of equalities processed by ProcessInequality. We will use $\_{i}$ to denote the value of variables at step i, and $\_^{j}$ to denote the value of variables during the j^{th} iteration of the loop at line 7. For the base case when no inequalities have been processed yet, $i = 0$ and val_0 and M_0 are the zero valuation that assigns 0 to everything. Therefore $\mathsf{val}_0 \preceq \mathsf{L}$ and $\mathsf{M}_0 \preceq \mathsf{L}$.

For the inductive step, we assume that $\mathsf{M}_i \preceq \mathsf{L}$ and $\mathsf{val}_i \preceq \mathsf{L}$ and show that after processing the $i + 1$ inequality this property still holds. We will prove this by induction on the loop iterations of the while loop at line 7 by showing that $\mathsf{val}_i^j \preceq \mathsf{L}$ and $\mathsf{M}_i^j \preceq \mathsf{L}$ is a loop-invariant.

Base case: When no loop iterations have been executed, $\mathsf{M}_i^0 = \mathsf{M}_i$. By inductive assumption $\mathsf{M}_i^0 \preceq \mathsf{L}$. Before the while-loop, $\mathsf{val}_i^0(x) = \mathsf{M}_i(x)$ for all $x \neq b$. The $\mathsf{newVal}(a \lhd b)$ procedure returns $\mathsf{M}_i(a)$ if $\lhd$ is $\leq$ and $\mathsf{M}_i(a) + 1$ otherwise. Since $\mathsf{M}_i \preceq \mathsf{L}$ (inductive hypothesis), $\mathsf{newVal}(a \lhd b)$ returns a lower bound on the value b can be assigned to in order to satisfy $a \lhd b$. Therefore, before the while-loop $\mathsf{val}_i^0 \preceq \mathsf{L}$.

Inductive step: Let M_i^j and val_i^j be the values of M_i and val_i at the beginning of loop iteration j. By the inductive hypothesis we know that $\mathsf{M}_i^j \preceq \mathsf{L}$ and $\mathsf{val}_i^j \preceq \mathsf{L}$. M_i^{j+1} only differs from M_i^j at x^j (updated at line 9), with $\mathsf{M}_i^{j+1}(x^j) = \mathsf{val}_i^j(x^j)$, if the $\mathsf{updateModel}$ call succeeded (if it failed the procedure terminates and M and val are not updated anymore). However $\mathsf{val}_i^j \preceq \mathsf{L}$ (inductive hypothesis) so $\mathsf{val}_i^j(x^j) \leq \mathsf{L}(x^j)$.

Similarly, val_i^{j+1} only differs from val_i^j at each y^j, where it is set to the maximum value of $\mathsf{val}_i^j(y^j)$ and $\mathsf{newVal}(x^j \lhd y^j)$ (line 13). We know that $\mathsf{val}_i^j(y^j) \leq \mathsf{L}(y^j)$ (inductive hypothesis), and that $\mathsf{newVal}(x^j \lhd y^j)$ will return a lower bound on the value of y^j that satisfies $x^j \lhd y^j$ assuming x^j has the value $\mathsf{M}_i^j(x^j)$. Therefore $\mathsf{val}_i^{j+1}(y^j) \leq \mathsf{L}(y^j)$ and $\mathsf{val}_i^{j+1} \preceq \mathsf{L}$.

□

Claim 4. The $\mathsf{updateModel}$ procedure only returns *true* if $\mathcal{I}$ is unsatisfiable.

Proof. The call to $\mathsf{updateModel}(x, \mathsf{val}(x))$ returns *true* only in the following cases:

1. x is a constant c such that $c < \mathsf{val}(x)$. Since we have already shown by induction that val provides a lower bound on the satisfying value of x ($\mathsf{val} \preceq \mathsf{L}$) it must be the case that $\mathcal{I}$ is *unsat*.

2. updating x to $\mathsf{val}(x)$ leads to an overflow. Similarly, because $val \preceq \mathsf{L}$.

3. $x = a$. If $\mathsf{updateModel}$ reaches a, there must be a cycle involving a. Since the $\mathsf{updateModel}$ procedure only gets called on $x \neq \mathsf{val}(x)$ and $\mathsf{val}(x) \preceq \mathsf{L}(x)$ the cycle must have at least one strict edge. Therefore $\mathcal{I}$ is *unsat*.

□

Claim 5. If $\mathcal{I}$ is satisfiable, the following is an invariant of the ProcessInequality procedure at line 7: $\forall x \notin \mathsf{PQ}$, $\mathsf{M}(x) = \mathsf{val}(x)$.

Proof. We prove this claim by induction on the number of while-loop iterations j.

Base case: For $j = 0$, no loop iterations have been executed. M is the same as val at all positions except b. However, if $\mathsf{val}(b) \neq \mathsf{M}(b)$, then b is added to PQ (line 6). Therefore the claim holds.

Inductive step: Assume all nodes x not in PQ at iteration j have $\mathsf{M}^j(x) = \mathsf{val}^j(x)$. We want to show that at the end of this iteration, $\mathsf{M}^{j+1}(x) = \mathsf{val}^{j+1}(x)$ for all x. Let x^j be the node popped from PQ during the $j + 1$ iteration. If $\mathsf{updateModel}$ succeeds, then $\mathsf{M}^{j+1}(x^j) = \mathsf{val}^{j+1}(x^j)$ (by line 9). By Claim 4, $\mathsf{updateModel}$ call cannot fail as by assumption, $\mathcal{I}$ is satisfiable. This is the only position where M^{j+1} differs from M^j, so $\forall x \notin \mathsf{PQ}, \mathsf{M}^{j+1}(x) = \mathsf{val}^j(x)$. Next, we check when val^{j+1} is updated. The val^{j+1} valuation is updated only at line 13 for all the successors y of x^j. So for all y' such that $x \lhd y' \notin \mathcal{I}$, $\mathsf{val}^{j+1}(y') = \mathsf{val}^j(y') = \mathsf{M}^{j+1}(y')$. However, for the values that do get updated, if $\mathsf{val}^{j+1}(y) \neq \mathsf{M}^{j+1}(y)$, y is added to PQ. Therefore for all $y \notin \mathsf{PQ}$, $\mathsf{val}^{j+1}(y) = \mathsf{val}^j(y) = \mathsf{M}^{j+1}(y)$.

□

Claim 6. If $\mathcal{I}$ is satisfiable, the algorithm will terminate with a satisfying valuation M.

Proof. We prove this by induction on the number of inequalities already processed by ProcessInequality. We will denote by $\mathcal{I}_i$ the set $\mathcal{I}$ corresponding to the i^{th} call of the ProcessInequality procedure. Base case: for $i = 0$, $\mathcal{I}_0 = \emptyset$ so the initial valuation M_0 is satisfying.

Inductive step: assume that the valuation M_i satisfies $\mathcal{I}_i$ and we are processing the $(i+1)^{th}$ inequality $a \lhd b$. We will show that at each loop iteration step j, the following invariant holds for all $x \lhd y \in \mathcal{I}_i$: $\mathsf{M}_i(x) \lhd \mathsf{val}_i(y)$.

Base case: $j = 0$, we have not executed any loop iterations yet. By inductive assumption, for all $x \lhd y \in \mathcal{I}_i$, $\mathsf{M}_i(x) \lhd \mathsf{M}_i(y)$. Before the loop iterations, val_i is the same as M_i for all terms except b. Updating $\mathsf{val}_i(b)$ to $\mathsf{newVal}(a \lhd b)$ ensures that for the new inequality $a \lhd b$, $\mathsf{M}_i(a) \lhd \mathsf{val}_i(b)$, by the definition of newVal. Next we check that this update does not break the invariant on any of the other inequalities. Any inequality of the form $a' \lhd b \in \mathcal{I}_i$ must have been satisfied by M_i: $\mathsf{M}_i(a') \lhd \mathsf{M}_i(b)$. Since by definition of newVal, $\mathsf{newVal}(a \lhd b) \geq \mathsf{M}_i(b)$, the $a' \lhd b$ inequalities also satisfy the invariant $\mathsf{M}_i(a') \lhd \mathsf{val}_i(b)$. Note that the update to b's value cannot violate the invariant on inequalities of the form $b \lhd a'$.

Inductive step: assume that the invariant holds at iteration step j: forall $x \lhd y \in \mathcal{I}_i$, $\mathsf{M}_i^j(x) \lhd \mathsf{val}_i^j(y)$. Let x^j be the value popped off the queue during the j^{th} loop iteration. M is only updated during the $\mathsf{updateModel}$ procedure: $\mathsf{M}_i^{j+1}(x^j) = \mathsf{val}_i^j(x^j)$. Assuming the previous valuation M_i^j satisfied all the inequalities, the only inequalities that can violate the invariant after this update are of the form: $x^j \lhd y'$. However, during the for-loop at line 12, $\mathsf{val}_i^{j+1}(y')$ is assigned to a value that satisfies $x^j \lhd y'$ (by definition of newVal). Again, any inequality of the form $y' \lhd z$ cannot violate the invariant after the update to val since $\mathsf{M}(y')$ is unchanged.

Therefore, at the end of the while-loop for all $x \lhd y \in \mathcal{I}_{i+1}$, $\mathsf{M}_{i+1}(x) \lhd \mathsf{val}_{i+1}(y)$. By Claim 5 and because the loop exiting condition is $\mathsf{PQ} = \emptyset$, $\mathsf{M}_{i+1}(x) = \mathsf{val}_{i+1}(x)$ for all x. Therefore for all inequalities $x \lhd y \in \mathcal{I}_{i+1}$, $\mathsf{M}_{i+1}(x) \lhd$

$\mathsf{M}_{i+1}(y)$ i.e. at the end of the procedure M is a satisfying model for $\mathcal{I}$.

□

Lemma 7. If $\mathcal{I}$ is satisfiable, the $\mathsf{BV_{ineq}}$ algorithm maintains a least valuation M.

Proof. We will show this claim by proving that at each step i, the algorithm maintains the least valuation for the set of inequalities $\mathcal{I}_i$. Assume the set of input inequalities $\mathcal{I}$ are satisfiable. The proof proceeds by induction on step i, where step i corresponds to processing the i^{th} inequality using Algorithm 13. We will denote by M_i the valuation at step i and by $\mathcal{I}_i$ the set of inequalities at step i.

Base case : For $i = 0$ and $\mathcal{I}_0 = \emptyset$ the starting valuation M is the least valuation as it assigns everything to 0.

Inductive step : Say we are processing a new inequality $a \lhd b$. Since $\mathcal{I}_{i+1}$ is satisfiable there must exist some least valuation L that satisfies it. As shown by Claim 3, $\mathsf{M}_i \preceq \mathsf{L}$ is an invariant for the while-loop at line 7. Therefore at the end of the while-loop $\mathsf{M}_{i+1} \preceq \mathsf{L}$. By Claim 6, M_{i+1} satisfies $\mathcal{I}_{i+1}$, therefore M_{i+1} must be a least valuation for $\mathcal{I}_{i+1}$.

□

Theorem 8. The $\mathsf{BV_{ineq}}$ algorithm is sound and complete for set of inequalities $\mathcal{I}$.

Proof. The algorithm is trivially terminating, because at each iteration we increase the current M, and there are finitely many valuations. If we reach line 16, the valuation M must be a least valuation of $\mathcal{I}$ (by Lemma 7), so $\mathcal{I}$ must be satisfiable. By Claim 4, we only reach line 11 when $\mathcal{I}$ is unsatisfiable. Therefore the $\mathsf{BV_{ineq}}$ algorithm is sound and complete. □

Lemma 9. Algorithm 13 only updates the valuation M once for each node.

Proof. Let $x^1, \ldots, x^n$ be the sequence of literals popped from PQ during the loop iterations. The corresponding PQ keys will be:

$$\langle \mathsf{O}(x^1), \mathsf{val}^1(x^1) \rangle \ldots \langle \mathsf{O}(x^n), \mathsf{val}^n(x^n) \rangle$$

where we denote by $\_^j$ the value of the variable at loop iteration j. Note that while val changes, O stays the same. We will define the following ordering on the elements in the sequence: $x^i \sqsubseteq x^j$ if $\mathsf{O}(x^i) < \mathsf{O}(x^j)$, or if $\mathsf{O}(x^i) = \mathsf{O}(x^j)$ and $\mathsf{val}^i(x^i) \geq \mathsf{val}^j(x^j)$. We will show that the sequence of nodes popped from PQ is increasing with respect to the $\sqsubseteq$ ordering. By line 12 any y pushed on the priority queue PQ has $\mathsf{O}(y) \geq \mathsf{O}(x^j)$ where x^j is the element popped from PQ at that iteration. Therefore, because we always pop the element with lowest O value, the only way the ordering may fail is if $\mathsf{O}(x^j) = \mathsf{O}(x^{j+1})$ and $\mathsf{val}^j(x^j) < \mathsf{val}^{j+1}(x^{j+1})$. We will show by induction that if $\mathsf{O}(x^j) = \mathsf{O}(x^{j+1})$, then it must be the case that $\mathsf{val}^j(x^j) \geq \mathsf{val}^{j+1}(x^{j+1})$.

Base case: For $j = 1$, there is only one element in the sequence so it respects the $\sqsubseteq$ ordering.

Inductive step: Assume that the $\sqsubseteq$ ordering holds up to index j. Let x^j be the element just popped off of PQ we are currently processing. We want to show that if $\mathsf{O}(x^j) = \mathsf{O}(x^{j+1})$ then $\mathsf{val}^j(x^j) \geq \mathsf{val}^{j+1}(x^{j+1})$, where x^{j+1} is the next element in the sequence. If x^{j+1} was already in PQ, then $\mathsf{val}^j(x^{j+1}) \leq \mathsf{val}^j(x^j)$ (by queue order). If $\mathsf{val}^{j+1}(x^{j+1}) = \mathsf{val}^j(x^{j+1})$ the property holds. The value of x^{j+1} may be updated at line 13 to either the old value $\mathsf{val}^j(x^{j+1})$, in which case it respects the ordering, or to $\mathsf{newVal}(x^j \lhd x^{j+1})$.

Assume $\lhd$ was $<$. If $x^j \lhd x^{j+1}$ is not the new inequality $a \lhd b$, $\mathsf{O}(x^j) < \mathsf{O}(x^{j+1})$ (Claim 6) so the property holds. If $x^j = a$, updateModel fails and the procedure terminates.

Now assume $\lhd$ was $\leq$. The newVal procedure will then return $\mathsf{M}^{j+1}(x)$ which is equal to $\mathsf{M}^{j+1}(x^j) = \mathsf{val}^j(x^j)$ (by line 9). Therefore $\mathsf{val}^{j+1}(x^{j+1}) = \mathsf{val}^j(x^j)$. Because $\sqsubseteq$ is transitive, the ordering holds for the sequence.

Assume there is some element x that gets updated more than once. Let $i < j$ be the first two positions it appears in the sequence at. Because the sequence is sorted w.r.t. $\sqsubseteq$ it must be the case that all the $\mathsf{O}(x^k)$ for $i \leq k \leq j$ are equal, and $\mathsf{val}^i(x) \geq \mathsf{val}^j(x)$. Consider time-step $j-1$ when we are just about to add x to PQ for the second time. By line 13 as the value of x is updated to the maximum between the old value $\mathsf{val}^i(x)$ and some other value, $\mathsf{val}^j(x) \geq \mathsf{val}^i(x)$. Before adding x to the queue we check whether $\mathsf{val}^j(x) \neq \mathsf{M}^j(x)$. However, the last time we updated $\mathsf{M}(x)$ was at time step i (by assumption), so $\mathsf{M}^j(x) = \mathsf{val}^i(x)$. Since by assumption we are adding x to the queue again, $\mathsf{val}^j(x) \neq \mathsf{val}^i(x)$. However, this leads to a contradiction because $\mathsf{val}^j(x) \geq \mathsf{val}^i(x)$, but due to the ordering $\mathsf{val}^i(x) \geq \mathsf{val}^j(x)$. ☐

Because each node only gets updated once (Lemma 9), the while-loop can run at most n times, where n is the number of nodes in the inequality graph. Let m be the number of inequalities in $\mathcal{I}$ and n the number of bit-vector terms. If a Fibonacci Heap is used to implement the priority queue PQ, each pop can be executed in $O(\log n)$, and the push operations in amortized $O(1)$. Since the for-loop at line 12 only visits each edge once, the push and updates to val will only be executed m times during the while-loop. Therefore the ProcessInequality procedure runs in $O(n \log n + m)$, which makes the full inequality algorithm run in $O(m(n \log n + m))$.

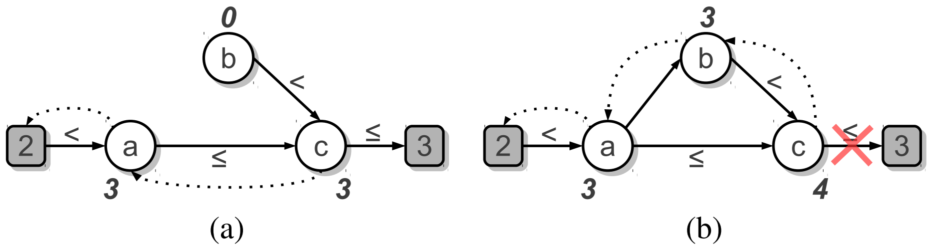

Figure 4.6: Inequality graph for Example 15.

Example 15. Consider the following set of inequalities over bit-vector terms of bit-width 8 where, for brevity, we use decimal numerals to denote bit-vector constants: $\mathcal{I} = \{2 < a, a \leq c, b < c, c \leq 3\}$. Figure 4.6a shows the least satisfying model for $\mathcal{I}$. The nodes are bit-vector terms; gray nodes are constants, and white ones, variables. Each node has an associated constant, its M value. The continuous edges represent inequalities. The dotted edges are *reason* edges: they point to the node that forced the last update to the current node's value.

To process the new inequality $a \leq b$, we add the corresponding inequality edge, and update the value of b to $\mathsf{M}(a)$. This in turn requires increasing the value of c to $\mathsf{M}(b) + 1$. We identify a conflict when $\mathsf{updateModel}(3, 4)$ fails: 3 is a constant and $M(c) \leq 3$ does not hold (Figure 4.6b). Because c has the lowest possible value, $\mathcal{I}$ must be unsatisfiable. We build the following minimal conflict by traversing the reason back-edges: $\{2 < a, a \leq b, b < c, c \leq 3\}$.

4.4 In-processing Solver

The lazy DPLL($\mathcal{T}$) framework enables several techniques that are difficult or impossible to use with eager solvers. In this section we discuss one of these techniques consisting of applying word-level rewrites during solving (*in-processing*).

Before engaging in potentially expensive SAT reasoning, $\mathsf{BV_{bb}}$ relies on the inprocessing module to check if the problem can be solved or significantly simplified by word-level simplification techniques. The algorithm described in Algorithm 15 has the flavor of Gaussian elimination. It works by iterating over a worklist of assertions W while maintaining a substitution map σ. Initially, W is initialized to the set of assignments A_{bv} active in the current search context and σ is empty. The worklist assertions first go through a preprocessing step that consists of slicing variables and introducing fresh Skolem variables for the slices. We do this by collecting the *cut-points* of bit-vector variables. A cut-point point at position i signifies a cut lying between indices $i - 1$ and i. For each variable v and extract $v[i : j]$ occurring in A_{bv} the i and $j + 1$ positions are marked as cut-points. There are implicit cuts at position 0 and n. For any two adjacent cuts $i, j + 1$ we introduce a fresh Skolem variable s for $v[i : j]$ and add the following to the substitution map:

$$v \leftarrow s_1 \circ \ldots \circ s_k.$$

Slicing variables in this manner enables more substitution and simplifications. Given $v[i : j] = t$, it is not sound to use t to replace $v[i : j]$ as it may overlap with other extract terms. If however $v[i : j]$ is a disjoint slice the substitution is safe.

The algorithm then begins the solving loop by iterating through the work-list W and applying substitutions. For each worklist assertion $w \in W$, we first apply the substitution map, and then rewrite it using word-level simplification techniques (simplify). The solveEq procedure then attempts to solve the updated assertion w to obtain a new substitution and learn new equalities entailed by w and add these to the working list. In our implementation, we solve xor equations and we slice equations between concat expressions to get new equalities. The working list W and the substitution map σ are updated with this new information, and the process is repeated to a fix-point.

If any of the assertions in W reduces to $false$, we have a conflict, and if they all reduced to *true* the assertions are satisfiable. If there are no such obvious inconsistencies we assert $\mathsf{bitblast}(\mathsf{W})$ to a SAT solver and run the Solve routine on the simplified set of assertions W. We do this heuristically, if the problem has been reduced enough in terms of the circuit size. We found checking the simplified assertions when they are less than 50% of the size of the original assertions to be a good heuristic.

All of the data-structures described in this section are enhanced with extra book-keeping information that allows for retrieving the conflict in terms of the original assertions. The substitution map labels each substitution with the original set of assertions that enabled it.

The call to Solve on line 14 in Algorithm 15 calls to another instance of the SAT solver. Each call resets the SAT solver, and starts from scratch. Bit-blasting the conjunction of simplified assertions W to $\mathrm{SAT}_{\mathsf{bb}}$ would not be sound: these assertions hold only during the current part of the search space. Bit-blasting $a \Leftrightarrow \bigwedge \mathsf{bitblast}(w)$ for $\mathcal{T}_{\mathsf{bv}}$-literals $w \in W$ using a fresh assumption literal a would be sound. However this has several drawbacks. First, bit-blasting each set of simplified assertions W corresponding to an assignment A_{bv} can lead to a different set of clauses. These clauses would accumulate and have the potential of slowing down the SAT solver. More importantly perhaps, the state of $\mathrm{SAT}_{\mathsf{bb}}$ has to be carefully maintained between calls to $\mathcal{T}_{\mathsf{bv}}$-check to be able to provide lazy explanations. The main benefit in reusing the same SAT solver between queries is taking advantage of the learned clauses in the previous queries. However, this is diminished by the introduction of fresh Skolem variables during slicing and the fact that the bit-blasted formula changes significantly from query to query due to the simplifications.

Example 16. Assume the $\mathsf{In\text{-}processing}$ solver is called on the following set of bit-vector assertions:

$$\mathsf{A}_{\mathsf{bv}} = \left[x = 257_{[16]}, y = x[15:8] \times x[7:0] + 7_{[16]}, y \neq 8\right]$$

The call to slice slices the x variable by introducing fresh Skolem variables $s_{[8]}$ and $s'_{[8]}$. The substitution map σ and the worklist assertions W are updated to:

$$\langle W, \sigma\rangle = \langle \left[s \circ s' = 257_{[16]}, y = s \times s' + 7_{[16]}, y \neq 8\right], \{x \leftarrow s \circ s'\}\rangle.$$

The $\mathsf{solveEq}$ procedure uses the $s \circ s' = 257_{[16]}$ equality to learn two new equalities:

$$\langle W, \sigma\rangle = \langle \left[s = 1_{[8]}, s' = 1_{[8]}, s \circ s' = 257_{[16]}, y = s \times s' + 7_{[16]}, y \neq 8\right], \{x \leftarrow s \circ s'\}\rangle.$$

This enables the following sequence of substitutions and simplifications (the assertions that simplify to *true* are removed):

$$\begin{aligned}
\langle W, \sigma\rangle &= \langle \left[s' = 1_{[8]}, 1_{[8]} \circ s' = 257_{[16]}, y = 1_{[16]} \times s' + 7_{[16]}, y \neq 8\right], \\
&\qquad \{x \leftarrow s \circ s', s \leftarrow 1_{[8]}\}\rangle \\
&= \langle [y = 8, y \neq 8], \{x \leftarrow s \circ s', s \leftarrow 1_{[8]}, s' \leftarrow 1_{[8]}\}\rangle \\
&= \langle [\bot], \{x \leftarrow s \circ s', s \leftarrow 1_{[8]}, s' \leftarrow 1_{[8]}, y \leftarrow 8\}\rangle
\end{aligned}$$

Experimental evaluation. We evaluated the impact of each algebraic technique on all the SMT-LIB v2.0 benchmarks in the $\mathsf{QF\_BV}$ family. Figure 4.7 shows the performance benefit of using the equality solver $\mathsf{BV}_{\mathsf{eq}}$, by comparing cvcLz with the equality solver enabled (cvcLz) with cvcLz with equality reasoning disabled ($\mathsf{cvcLz\text{-}Eq}$). Disabling the

Algorithm 15: In-processing.

```
Input: $A_{bv}$
$\langle W, \sigma \rangle \leftarrow \langle A_{bv}, [] \rangle$;
$\langle W, \sigma \rangle \leftarrow$ slice($A_{bv}$, $\sigma$);
changed $\leftarrow$ true;
while changed do
    changed $\leftarrow$ false;
    for $w \in W$ do
        $w \leftarrow$ simplify($\sigma(w)$) ;
        $\langle W', \sigma' \rangle \leftarrow$ solveEq(w);
        if $W' \neq \emptyset$ or $\sigma \neq []$ then
            changed $\leftarrow$ true ;
        $\langle W, \sigma \rangle \leftarrow \langle W \cup W', \sigma; \sigma' \rangle$;
if false $\in W$ then
    return conflict;
return Solve(bitblastW);
```

equality solver leads to fewer problems solved in more time. However, enabling the core solver that relies on slicing to be complete hurts performance overall (Figure 4.8). The only exception is the bruttomesso family of benchmarks crafted to stress test core solvers.

Figure 4.9 shows the performance impact of disabling the inequality solver BV_{ineq}. While in most cases the inequality solver helps performance, there is a clear line above the diagonal consisting of the benchmarks where inequality reasoning did not help but added an small overhead. As Figure 4.10 shows, overall the algebraic inprocessing module helps performance. There are, however, several problems where performance is worse. We attribute this to the overhead of the SAT reasoning on the simplified assertions. While we try to account for this by disabling SAT solving in the in-processing module dynamically if it has not been successful on previous assertions, there are still instances where the overhead outweighs the benefit.

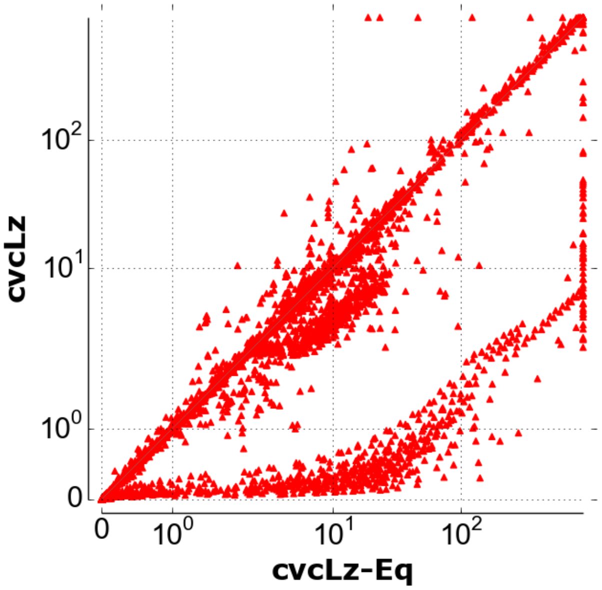

Figure 4.7: Disabling equality sub-solver (cvcLz-Eq).

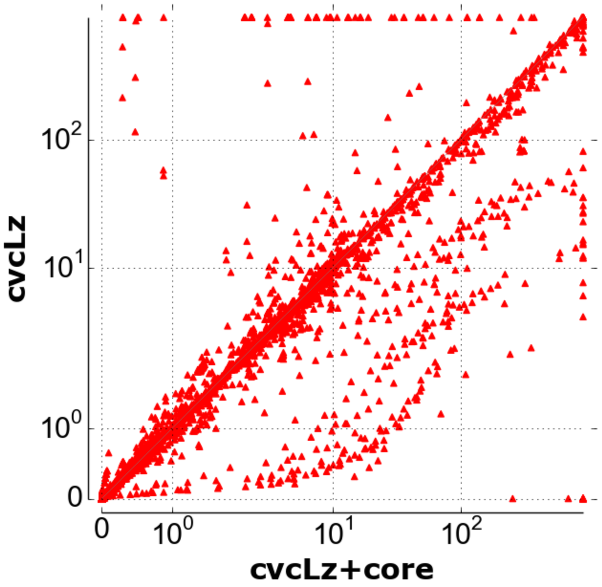

Figure 4.8: Enabling core solver (cvcLz+core).

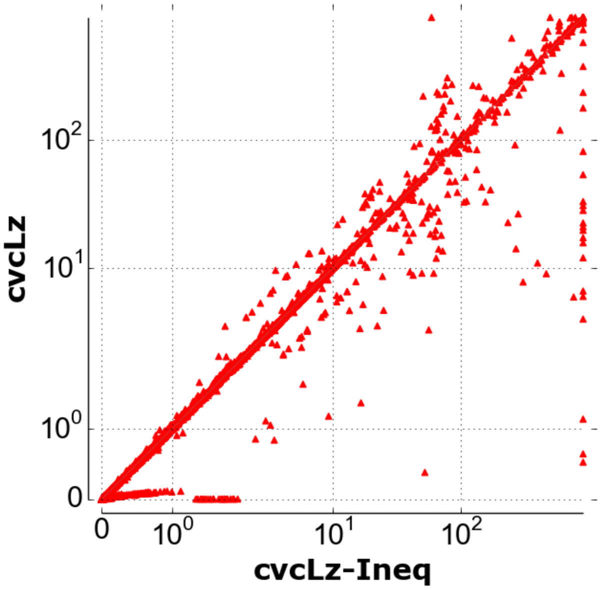

Figure 4.9: Disabling inequality sub-solver (cvcLz-Ineq).

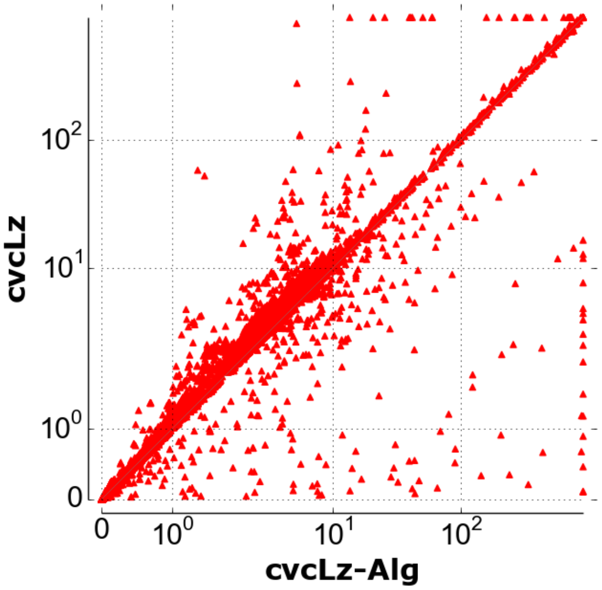

Figure 4.10: Disabling algebraic in-processing (cvcLz-Alg).

4.5 Lazy Techniques

Integrating a lazy $\mathcal{T}_{\mathsf{bv}}$ solver in the CDCL($\mathcal{T}$) framework allows for using certain techniques that would not otherwise be easily integrated with an eager solver. This section highlights some of these techniques.

Lemmas for Bit-vectors. Certain arithmetic proprieties are hard for SAT solvers to reason about, especially when they involve multiplication or division. Some of these properties can be axiomatized via lemmas instantiated to values specific to the problem at hand. The splitting on demand infrastructure allows doing this on demand, only when the lemma is relevant in the current search context. One such lemma our solver LBV employs encodes a basic property of division: the remainder is always smaller than the divisor:

$$\forall x.\forall r.\forall d.\forall x.(x\%d = r \Rightarrow (d = 0_{[n]} \vee r < d)).$$

Not only is this lemma quantified, but it is bit-width independent. Therefore it must be instantiated for specific problems. When an assertion of the form $t_1\%t_2 = t_3$ appears in A_{bv}, the lemma is instantiated with t_1, t_2 and $t3$ and the following clause is added to $\text{SAT}_{\mathsf{main}}$:

$$\neg(t_1\%t_2 = t_3) \vee (t_2 = 0_{[n]}) \vee (t_3 < t_2).$$

Conflict Minimization The conflicts computed by $\mathsf{BV}_{\mathsf{bb}}$ using SAT_{bb} with assumptions are non-minimal and so are the conflicts computed by $\mathsf{BV}_{\mathsf{alg}}$. We noticed that in some instances they are far from minimal. The average number of literals in conflicts returned for over 2500 QF_BV benchmarks could be reduced to 75% of the original size. In several instances the the conflicts could be reduced to 10% of the original size. For this reason we implemented a conflict minimization scheme based on the quickXplain conflict minimization algorithm [53].

Figure 16 shows the pseudo-code for the recursive minConflict procedure. The two arguments to minConflict are confl, the conflict to be minimized and minC an argument passed by reference in which the minimized conflict is stored. Initially $\mathsf{minC} = \emptyset$. The procedure employs an incremental SAT solver that maintains an assertion stack on which assertions are pushed using push and popped using pop. Our implementation uses solve with assumptions to simulate incrementality. The SAT solver state is maintained during recursive calls: the un-popped assertions act as background assertions. The conflict confl is minimized with respect to these background assertions.

The procedure employs a binary search strategy to find an unsatisfiable subset of confl. If confl has only one element, the element is added to minC. Otherwise, the routine checks if the top half of the conflict confl top(confl) is unsatisfiable. If this is the case unsatCore attempts to further minimize this by identifying a subset of top(confl) that is unsatisfiable (it relies on the assumptions conflict procedure in Algorithm 9). This new subset is recursively minimized.

Next, the algorithm applies the same method if the bottom half bottom(confl) is unsatisfiable. If neither the top or bottom halves of confl are unsatisfiable, it must be that the minimal conflict contains literals in both sides. Before the recursive call at line 19 the literals in bottom(confl) are still asserted. The minConflict call will add to minC minimum subset of the literals in top(confl) that along with bottom(confl) are unsatisfiable. The recursive call at line 23 will minimize the literals in bottom with respect the literals in the minimized conflict computed so far, including those selected from top.

Because the satSolve routine could be very expensive, our implementation runs the SAT solver with a $10K$ bound on the number of conflicts. Therefore, satSolve could return *unknown*, in which case we conservatively act as if the result was *sat*.

Algorithm 16: Conflict minimization minConflict based on quickXplain.

```
Input: ⟨confl, minC⟩
if |confl| = 1 then
    minC ← minC ∪ confl;
    return;
push ( );
assert ( top ( confl ) );
if Solve ( ) = unsat then
    confl'←unsatCore ( top ( confl ) );
    pop ( );
    minConflict ( confl', minC );
    return;
pop ( );
push ( );
assert ( bottom ( confl ) );
if Solve ( ) = unsat then
    confl'←unsatCore ( bottom ( confl ) );
    pop ( );
    minConflict ( confl', minC );
    return;
minConflict ( top ( confl ), minC );
pop ( );
push ( );
assert ( minC );
minConflict ( bottom ( confl ), minC );
pop ( );
```

Justification heuristic Section 2.3 described the use of a non-clausal engine, SAT$_{\mathcal{J}}$ to reduce the size of the problem theory solvers have to reason about. This feature proves very beneficial for the performance of the lazy bit-vector solver cvcLz.

Experimental evaluation. Figure 4.11 shows the performance impact of cvcLz with and without SAT$_{\mathcal{J}}$ (cvcLz-J). While there are some problems on which performance is

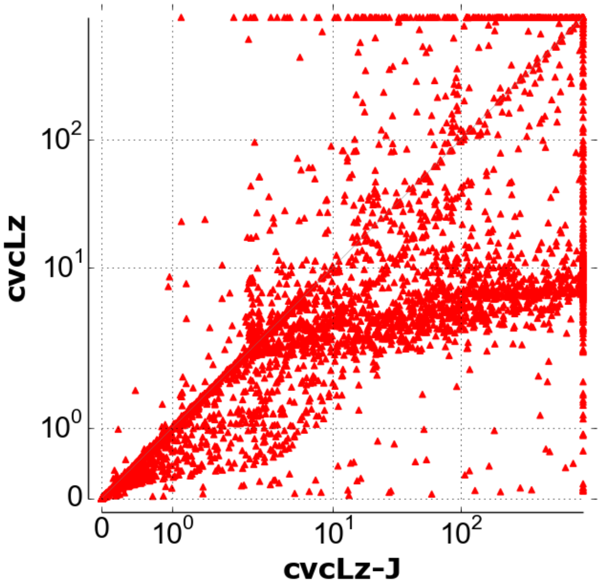

Figure 4.11: Disabling justification heuristic (cvcLz-J).

worse since the justification heuristic changes the SAT search, overall the performance is improved. Enabling $\mathrm{SAT}_{\mathcal{J}}$ solves 175 more problems in half the total time.

Figure 4.12 compares the average number of literals in the $\mathcal{T}_{\mathsf{bv}}$ conflicts with and without quickXplain enabled. Note that returning different conflicts affects the search, so the values shown here are not equivalent to how much quickXplain reduced the conflict size. They do illustrate that using quickXplain greatly reduces the conflict size.

The results in Figure 4.13 show the performance impact of this routine. Although the conflicts are minimized by a factor of over 10 in some instances, the overhead of minimizing the conflicts is far too large. The quickXplain algorithm requires only $O(n \log(k+1) + k^2)$ [53] checks, where k is the size of the minimized conflict and n of the initial conflict, but each one of these checks can be quite expensive. This area requires further investigation, such as a more efficient conflict minimization procedure.

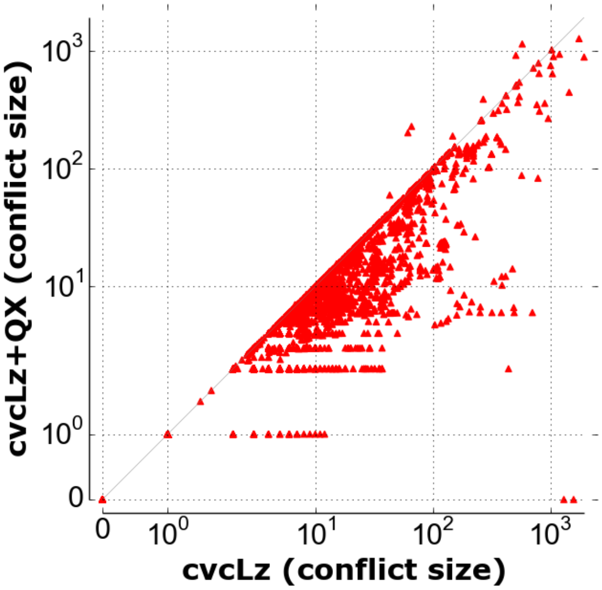

Figure 4.12: Average conflict size using QuickXplain(cvcLz+QX).

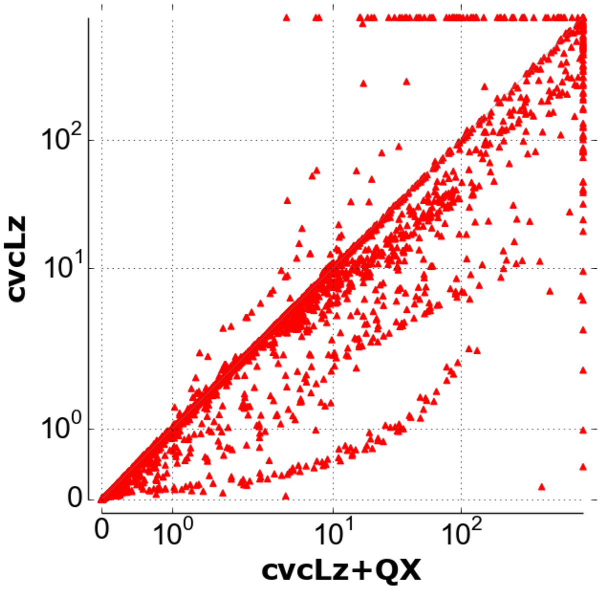

Figure 4.13: QuickXplain conflict minimization (cvcLz+QX).

4.6 Model Generation

An important feature of an SMT solver is being able to return a set of values that satisfy the input formula, if such a set of values exists. These are often used as counter-examples. An SMT *model* for a satisfiable $\mathcal{T}$-formula ψ is a satisfying assignment from the free variables in ψ to constants in their respective domains. Note that this is not the same notion of model introduced in Section 1.2.2.[9]

In this section we give a brief overview of how models are generated in cvcLz. We will assume the CVC4 $\mathcal{T}$-independent preprocessing does the book-keeping necessary to build a model from a model of the preprocessed formula.

The LBV solver must be able to return a model that satisfies the satisfiable $\mathcal{T}_{\mathsf{bv}}$ assertions A_{bv}. Note that the value of variables not occuring in A_{bv} is not relevant and can be set to an arbitrary value such as $0_{[n]}$. LBV requests a model from the last sub-theory that was complete. Recall that if the equality sub-solver $\mathsf{BV}_{\mathsf{eq}}$ is complete, it attempts to build a model by assigning distinct values for each congruence class. If queried, it returns this model. The inequality sub-solver $\mathsf{BV}_{\mathsf{ineq}}$ always maintains a least satisfying valuation which can be used to build a model. The bit-blasting solver $\mathsf{BV}_{\mathsf{bb}}$ has to query the $\mathrm{SAT}_{\mathsf{bb}}$ for the value of the bits of the variables occuring in A_{bv}. The inprocessing module retrieves the value of a variable v by first applying the substitution map σ to v and then collecting the values of all the variables in $\sigma(v)$ from the SAT solver it bit-blasted to.

---

[9] In SMT, free variables are regarded as uninterpreted constants so they are actually part of the model.

4.7 Related Work

The way the bit-blasting SAT solver $\mathsf{BV}_{\mathsf{bb}}$ integrates two SAT solvers is similar to the work on IC3 in [12]. They show a similar way of composing SAT solvers that allows unit propagation between the SAT solvers, and show the performance benefits of integrating such an approach in IC3.

Word-level approaches to solving bit-vector constraints avoid reduction to SAT and attempt to employ word-level reasoning. The work in [30] and [11] is based on word-level reasoning and it uses Shostak-style canonizers and solvers to compute a canonical form for bit-vector expressions. While an elegant approach, its applicability is limited to a restricted set of operators: concatenation, extraction and linear equations over bit-vectors.

The framework for a lazy bit-vector solver was first introduced by Bruttomesso et al. [23]. They describe an implementation of a DPLL($\mathcal{T}$)-style lazy layered solver for $\mathcal{T}_{\mathsf{bv}}$ in the SMT solver MathSAT [27]. Their approach lazily encodes the problem into linear integer constraints and uses word-level inference rules during solving. Later work by Franzen [42] moves from encoding the problem into linear integer arithmetic to bit-blasting the formula to the same SAT solver used to reason about the Boolean abstraction of the formula.

The lazy solver cvcLz described in this chapter extends the lazy CDCL-style approach to bit-vector solving in [27]. We explored the following new ideas within the lazy framework: (i) a dedicated SAT solver for $\mathcal{T}_{\mathsf{bv}}$ that supports bit-blasting-based propagation with lazy explanations; (ii) specialized $\mathcal{T}_{\mathsf{bv}}$ sub-solvers that reason about fragments of $\mathcal{T}_{\mathsf{bv}}$; (iii) inprocessing techniques to reduce the size of the bit-blasted formula when possible; and (iv) decision heuristics to minimize the number of literals sent to the bit-vector solver by the main SAT engine.

These new features greatly improve performance: our solver solves 450 more problems in roughly one third of the time compared to the only other lazy bit-vector solver. This brings the lazy framework from a niche player to a serious contender.

Chapter 5

Lazy vs Eager

So far we have presented two bit-vector solvers employing different approaches to solving bit-vector constraints: the eager solver `cvcE` (Chapter 3) and the lazy solver `cvcLz` (Chapter 4). This chapter provides a comparative analysis of the two solvers. We try to answer questions such as: are there types of problems for which one particular approach is better suited for and if yes, why? We start by providing an extensive experimental evaluation comparing the performance of the lazy and eager approaches in Section 5.1. This section also compares the performance of the CVC4 bit-vector solvers with that of other state-of-the-art bit-vector solvers. In Section 5.2, we look in more depth at the reason for the performance difference between the two solvers, while focusing on specific problem types. Finally, in Section 5.3 we conclude by discussing other approaches to solving bit-vector constraints.

5.1 Experimental evaluation

In this section, we present a comparative experimental evaluation of the eager (cvcE) and lazy (cvcLz) approaches as implemented in the SMT solver CVC4. All the experiments in this section were run on the StarExec [80] cluster infrastructure with a timeout of 900 seconds and a memory limit of 50GB.[1] For the QF_BV experiments, we included all the SMT-LIB 2014-06-03 benchmarks. The QF_AUFBV experiments contain all the SMT-LIB 2014-06-03 benchmarks for the QF_ABV and QF_AUFBV logics.

Table 5.1 compares the performance of cvcE, cvcLz and that of the only other bit-vector solver that supports lazy bit-blasting, mathsatLz[2]. The eager solver performs better on families that involve bit-level manipulations, such as the brummayerebiere* families, the asp family that does not contain any arithmetic operators, and the float family.

The lazy solver cvcLz excels on families that benefit from algebraic reasoning, such as calypto, tacas07, bruttomesso and uclid_contrib_smtcomp09. Overall, the lazy solver cvcLz solves more problems than the eager solver cvcE in less time. Compared to the only other lazy solver we are aware of, mathsatLz, cvcLz solves 434 more problems, in one third of the time.

Complementary approaches. A closer examination of the results in Table 5.1, shows that there are few families on which the two solvers cvcE and cvcLz perform similarly: on most families one greatly outperforms the other. Figure 5.1 shows a scatter plot of the run-time of cvcLz compared to that of cvcE (note that this plot is not on a log scale). It is

---

[1] Experiments were run on the queue all.q consisting of Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz machines with 268 GB of memory.

[2] We used the SMTCOMP2014 submission and altered the configuration script to enable the lazy solver.

hard to draw any conclusions from this image, as the results vary wildly by benchmark. If we eliminate the 709 benchmarks in the asp and log-slicing families, an interesting pattern emerges (Figure 5.2). The lack of problems lying on the diagonal suggests the two approaches are complementary: the lazy solver efficiently solves problems that are either impossible or very difficult for eager solvers. At the same time, it is not realistic to expect the lazy solver to do well on problems that are easy for eager solvers (and indeed it is often slower on these problems).

For this reason we propose a portfolio approach that runs an eager solver and a lazy solver in parallel. To gauge the complementary nature of the two approaches we used CVC4's portfolio infrastructure which allows us to run the two solvers in different parallel threads. In this setup, the solver waits for the first thread that finishes with an answer and then kills the other, thus getting the best performance between the two solvers each time (modulo memory usage). We use cvcPll to refer to the parallel solver implementation in CVC4, and cvcVBS for the *virtual best solver* from combining cvcE and cvcLz. Table 5.2 compares cvcPll with cvcVBS and the lazy and eager solvers. It shows that cvcPll is very close to cvcVBS in terms of performance. Most of the problems that are not solved by cvcPll but are solved by cvcVBS are in the asp and log-slicing families that use a lot of memory for solving. Combining the two solvers dramatically increases the number of problems solved in both cvcPll and cvcVBS.

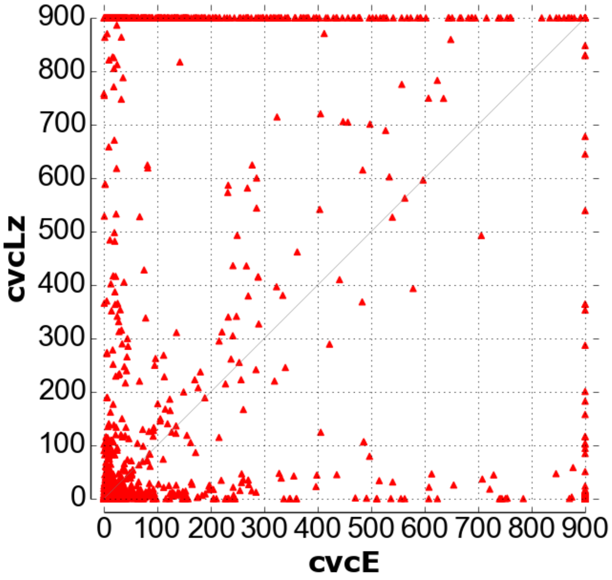

Figure 5.1: Comparing cvcLz with cvcE on all of QF_BV.

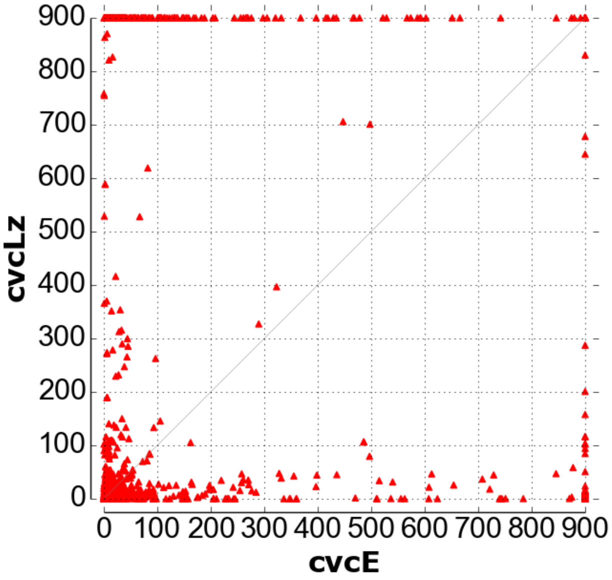

Figure 5.2: Comparing cvcLz with cvcE on subset of QF_BV (excluding asp and log-slicing.

Table 5.1: Comparing cvcLz and cvcE on QF_BV.

	cvcLz		cvcE		mathsatLz	
set	solved	time (s)	solved	time (s)	solved	time (s)
RWS (20)	16	202.71	**18**	**320.99**	16	210.35
VS3 (11)	**0**	**0.0**	0	0.0	0	0.0
asp (501)	146	27661.7	**303**	**45233.95**	155	42202.8
bench_ab (285)	**285**	**1.97**	285	15.31	285	3.61
bmc-bv (135)	132	189.23	**135**	**503.52**	132	313.14
bmc-bv-svcomp14 (66)	63	1905.76	64	1388.82	**66**	**5042.46**
brummayerbiere (52)	37	708.96	**38**	**1978.26**	33	1877.61
brummayerbiere2 (65)	56	4110.15	**62**	**1495.75**	62	3683.16
brummayerbiere3 (79)	22	2255.94	**45**	**3676.26**	27	5128.77
brummayerbiere4 (10)	**10**	**0.04**	9	0.46	6	3709.88
bruttomesso (976)	**976**	**2259.75**	421	27987.48	859	67305.78
calypto (23)	**17**	**1463.19**	12	951.24	11	187.28
challenge (2)	**0**	**0.0**	0	0.0	0	0.0
check2 (6)	**6**	**0.03**	6	0.31	5	0.06
crafted (21)	**21**	**0.14**	21	1.11	21	0.27
dwp_formulas (332)	**332**	**4.63**	332	18.51	332	6.37
ecc (8)	8	4.35	8	11.96	**8**	**1.48**
fft (23)	2	760.96	**7**	**182.07**	2	17.1
float (213)	51	4654.23	**163**	**14940.92**	79	8831.68
galois (4)	1	0.33	**1**	**0.26**	1	1.53
gulwani-pldi08 (6)	6	26.4	**6**	**23.55**	6	29.14
log-slicing (208)	59	22441.22	**69**	**22249.99**	22	10548.76
mcm (186)	21	4245.53	**82**	**7643.94**	4	1302.3
pipe (1)	**0**	**0.0**	0	0.0	0	0.0
pspace (86)	**86**	**0.4**	86	378.89	10	5644.74
rubik (7)	4	546.87	6	617.93	**7**	**540.87**
sage (26607)	26607	13324.64	26607	14436.03	**26607**	**4121.58**
spear (1695)	1656	13874.81	**1690**	**24835.27**	1431	128710.43
stp (1)	**1**	**84.63**	0	0.0	1	567.98
stp_samples (426)	424	64.84	424	84.45	**424**	**14.57**
tacas07 (5)	5	148.38	5	945.13	**5**	**86.45**
uclid (416)	416	1250.4	416	2442.15	**416**	**512.92**
uclid_contrib (7)	**7**	**1061.88**	0	0.0	5	802.46
uum (8)	1	1.13	**2**	**26.28**	2	661.87
wienand-cav2008 (18)	14	18.4	**14**	**15.71**	14	21.57
	31488	**103273.59**	31337	172406.5	31054	292088.97

Table 5.2: Comparing cvcPll with cvcVBS, cvcLz and cvcE.

	cvcPll		cvcVBS		cvcLz		cvcE	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
RWS (20)	18	353.8	18	320.86	16	202.71	18	320.99
VS3 (11)	0	0.0	0	0.0	0	0.0	0	0.0
asp (501)	274	38527.8	312	43708.81	146	27661.7	303	45233.95
bench_ab (285)	285	4.37	285	1.97	285	1.97	285	15.31
bmc-bv (135)	135	156.85	135	226.11	132	189.23	135	503.52
bmc-bv-svcomp14 (66)	63	1343.17	65	1915.13	63	1905.76	64	1388.82
brummayerbiere (52)	39	634.81	39	585.92	37	708.96	38	1978.26
brummayerbiere2 (65)	62	1695.02	63	2266.06	56	4110.15	62	1495.75
brummayerbiere3 (79)	44	2367.93	46	3623.13	22	2255.94	45	3676.26
brummayerbiere4 (10)	10	0.12	10	0.04	10	0.04	9	0.46
bruttomesso (976)	976	2581.39	976	2259.74	976	2259.75	421	27987.48
calypto (23)	17	1181.85	17	970.31	17	1463.19	12	951.24
challenge (2)	0	0.0	0	0.0	0	0.0	0	0.0
check2 (6)	6	0.09	6	0.03	6	0.03	6	0.31
crafted (21)	21	0.32	21	0.14	21	0.14	21	1.11
dwp_formulas (332)	332	7.9	332	4.63	332	4.63	332	18.51
ecc (8)	8	5.63	8	4.35	8	4.35	8	11.96
fft (23)	7	206.12	7	182.07	2	760.96	7	182.07
float (213)	161	12807.95	164	13905.55	51	4654.23	163	14940.92
galois (4)	1	0.27	1	0.26	1	0.33	1	0.26
gulwani-pldi08 (6)	6	18.41	6	21.45	6	26.4	6	23.55
log-slicing (208)	67	22741.31	71	22772.67	59	22441.22	69	22249.99
mcm (186)	82	7424.03	82	7641.69	21	4245.53	82	7643.94
pipe (1)	0	0.0	0	0.0	0	0.0	0	0.0
pspace (86)	86	1.12	86	0.4	86	0.4	86	378.89
rubik (7)	6	730.22	6	617.93	4	546.87	6	617.93
sage (26607)	26606	14834.09	26607	12652.48	26607	13324.64	26607	14436.03
spear (1695)	1692	11618.22	1692	10481.83	1656	13874.81	1690	24835.27
stp (1)	1	113.49	1	84.63	1	84.63	0	0.0
stp_samples (426)	424	77.5	424	64.81	424	64.84	424	84.45
tacas07 (5)	5	153.67	5	127.65	5	148.38	5	945.13
uclid (416)	416	1457.69	416	1235.1	416	1250.4	416	2442.15
uclid_contrib_smtcomp09 (7)	7	1313.28	7	1061.88	7	1061.88	0	0.0
uum (8)	2	27.63	2	26.28	1	1.13	2	26.28
wienand-cav2008 (18)	14	16.81	14	15.09	14	18.4	14	15.71
	31873	122402.88	31924	126778.97	31488	103273.59	31337	172406.5

Table 5.3: Comparing cvcPll with other solvers on QF_BV.

	cvcPll		sonolar		yices		boolector		stp2		z3		4simp		mathsatE	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
RWS (20)	18	353.8	18	1534.19	16	126.34	**19**	**1542.5**	18	561.81	16	117.14	18	311.02	16	179.68
VS3 (11)	0	0.0	2	88.32	1	386.32	4	1500.33	3	661.29	**5**	**1217.04**	3	501.16	0	0.0
asp (501)	274	38527.8	377	51731.42	367	46627.97	385	36942.6	373	44788.27	**402**	**32453.66**	366	40628.41	282	44254.23
bench_ab (285)	285	4.37	285	0.66	**285**	**0.43**	285	0.77	285	4.61	285	1.87	285	1.87	285	3.6
bmc-bv (135)	135	156.85	**135**	**28.7**	135	45.97	135	891.04	135	57.95	135	84.8	135	47.43	135	219.85
bmc-bv-svcomp14 (66)	63	1343.17	64	865.84	64	685.67	64	476.44	64	301.34	64	556.6	64	312.6	**66**	**1422.53**
brummayerbiere (52)	39	634.81	41	487.25	41	778.48	41	1020.0	**41**	**392.3**	39	322.86	41	530.64	36	1879.49
brummayerbiere2 (65)	**62**	**1695.02**	57	4092.26	56	2679.68	62	2292.39	54	3709.58	41	4988.99	58	4173.07	61	2365.42
brummayerbiere3 (79)	44	2367.93	63	2944.4	34	3726.64	56	2019.54	**64**	**3364.4**	48	5336.69	61	4053.48	44	4859.04
brummayerbiere4 (10)	10	0.12	2	917.87	10	777.06	**10**	**0.02**	10	0.08	10	0.06	6	3215.25	0	0.0
bruttomesso (976)	**976**	**2581.39**	636	18255.05	648	33319.92	972	16869.39	971	76058.04	975	9854.81	717	18468.16	935	44553.62
calypto (23)	**17**	**1181.85**	13	335.7	9	1.26	12	406.27	13	578.7	13	876.06	11	358.38	13	516.53
challenge (2)	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	**1**	**754.29**	0	0.0
check2 (6)	6	0.09	6	0.01	**6**	**0.01**	5	0.01	6	0.06	6	0.03	6	0.02	5	0.06
crafted (21)	21	0.32	21	0.04	**21**	**0.03**	21	0.04	21	0.27	21	0.12	21	0.11	21	0.26
dwp_formulas (332)	332	7.9	332	0.98	**332**	**0.62**	332	1.17	332	4.29	332	2.3	332	2.63	332	5.2
ecc (8)	8	5.63	8	1.39	**8**	**0.24**	8	0.75	8	1.13	8	0.94	8	0.97	8	1.37
fft (23)	7	206.12	9	295.85	9	1001.6	9	553.97	9	477.79	9	741.81	**10**	**1075.39**	9	1642.44
float (213)	161	12807.95	**185**	**12735.48**	167	11040.74	169	20200.22	159	18433.63	146	16102.61	181	12596.02	165	13518.97
galois (4)	1	0.27	1	0.15	1	0.11	1	0.33	1	0.17	1	0.14	**1**	**0.07**	1	0.37
gulwani-pldi08 (6)	6	18.41	6	23.45	6	22.29	6	48.73	6	24.0	**6**	**15.67**	6	18.48	6	33.01
log-slicing (208)	67	22741.31	125	37046.76	13	6128.15	**134**	**20096.46**	101	23240.22	45	14738.41	122	30723.24	27	10533.6
mcm (186)	82	7424.03	**113**	**17001.44**	96	18961.5	61	10812.81	67	13448.24	84	14521.46	78	10168.71	68	11548.66
pipe (1)	0	0.0	0	0.0	0	0.0	1	47.77	1	70.03	0	0.0	**1**	**29.13**	0	0.0
pspace (86)	**86**	**1.12**	73	17121.45	21	2444.97	80	12115.3	42	60.66	9	4435.97	62	12832.25	55	19321.93
rubik (7)	6	730.22	7	222.01	7	333.27	7	730.67	6	131.24	6	73.74	**7**	**153.49**	7	259.9
sage (26607)	26606	14834.09	26607	3345.83	**26607**	**1141.17**	26607	4999.07	26607	3367.8	26607	3747.98	26607	2912.67	26607	4432.5
spear (1695)	1692	11618.22	1689	4479.0	**1695**	**519.28**	1691	32950.37	1695	2828.45	1686	7530.68	1695	6299.7	1688	132297.38
stp (1)	1	113.49	1	22.93	**1**	**3.1**	1	12.23	1	15.72	1	20.78	1	16.0	0	0.0
stp_samples (426)	424	77.5	424	10.48	**424**	**2.34**	424	8.04	424	16.24	424	10.69	424	12.78	424	14.35
tacas07 (5)	5	153.67	5	110.0	5	997.83	5	261.56	5	582.51	5	443.46	5	99.2	**5**	**22.47**
uclid (416)	416	1457.69	416	307.4	**416**	**43.0**	416	773.83	416	153.92	416	425.43	416	213.02	416	480.3
uclid_contrib (7)	7	1313.28	7	188.35	7	2580.49	7	1505.94	7	1342.59	7	1188.97	**7**	**176.04**	7	212.71
uum (8)	2	27.63	**2**	**6.99**	2	28.07	2	8.79	2	17.58	2	8.46	2	9.65	2	36.44
wienand-cav2008 (18)	**14**	**16.81**	9	31.11	14	34.38	14	19.8	14	32.18	14	25.4	4	20.67	14	30.77
	31873	122402.88	31739	174232.78	31524	134438.95	**32046**	**169109.18**	31961	194727.1	31868	119845.65	31762	150716.02	31740	294646.69

Table 5.4: Comparing cvcVBS with other solvers on QF_BV.

	cvcVBS		sonolar		yices		boolector		stp2		z3		4simp		mathsatE	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
RWS (20)	18	320.86	18	1534.19	16	126.34	**19**	**1542.5**	18	561.81	16	117.14	18	311.02	16	179.68
VS3 (11)	0	0.0	2	88.32	1	386.32	4	1500.33	3	661.29	**5**	**1217.04**	3	501.16	0	0.0
asp (501)	312	43708.81	377	51731.42	367	46627.97	385	36942.6	373	44788.27	**402**	**32453.66**	366	40628.41	282	44254.23
bench_ab (285)	285	1.97	285	0.66	**285**	**0.43**	285	0.77	285	4.61	285	1.87	285	1.87	285	3.6
bmc-bv (135)	135	226.11	**135**	**28.7**	135	45.97	135	891.04	135	57.95	135	84.8	135	47.43	135	219.85
bmc-bv-svcomp14 (66)	65	1915.13	64	865.84	64	685.67	64	476.44	64	301.34	64	556.6	64	312.6	**66**	**1422.53**
brummayerbiere (52)	39	585.92	41	487.25	41	778.48	41	1020.0	**41**	**392.3**	39	322.86	41	530.64	36	1879.49
brummayerbiere2 (65)	**63**	**2266.06**	57	4092.26	56	2679.68	62	2292.39	54	3709.58	41	4988.99	58	4173.07	61	2365.42
brummayerbiere3 (79)	46	3623.13	63	2944.4	34	3726.64	56	2019.54	**64**	**3364.4**	48	5336.69	61	4053.48	44	4859.04
brummayerbiere4 (10)	10	0.04	2	917.87	10	777.06	**10**	**0.02**	10	0.08	10	0.06	6	3215.25	0	0.0
bruttomesso (976)	**976**	**2259.74**	636	18255.05	648	33319.92	972	16869.39	971	76058.04	975	9854.81	717	18468.16	935	44553.62
calypto (23)	**17**	**970.31**	13	335.7	9	1.26	12	406.27	13	578.7	13	876.06	11	358.38	13	516.53
challenge (2)	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	**1**	**754.29**	0	0.0
check2 (6)	6	0.03	6	0.01	**6**	**0.01**	5	0.01	6	0.06	6	0.03	6	0.02	5	0.06
crafted (21)	21	0.14	21	0.04	**21**	**0.03**	21	0.04	21	0.27	21	0.12	21	0.11	21	0.26
dwp_formulas (332)	332	4.63	332	0.98	**332**	**0.62**	332	1.17	332	4.29	332	2.3	332	2.63	332	5.2
ecc (8)	8	4.35	8	1.39	**8**	**0.24**	8	0.75	8	1.13	8	0.94	8	0.97	8	1.37
fft (23)	7	182.07	9	295.85	9	1001.6	9	553.97	9	477.79	9	741.81	**10**	**1075.39**	9	1642.44
float (213)	164	13905.55	**185**	**12735.48**	167	11040.74	169	20200.22	159	18433.63	146	16102.61	181	12596.02	165	13518.97
galois (4)	1	0.26	1	0.15	1	0.11	1	0.33	1	0.17	1	0.14	**1**	**0.07**	1	0.37
gulwani-pldi08 (6)	6	21.45	6	23.45	6	22.29	6	48.73	6	24.0	**6**	**15.67**	6	18.48	6	33.01
log-slicing (208)	71	22772.67	125	37046.76	13	6128.15	**134**	**20096.46**	101	23240.22	45	14738.41	122	30723.24	27	10533.6
mcm (186)	82	7641.69	**113**	**17001.44**	96	18961.5	61	10812.81	67	13448.24	84	14521.46	78	10168.71	68	11548.66
pipe (1)	0	0.0	0	0.0	0	0.0	1	47.77	1	70.03	0	0.0	**1**	**29.13**	0	0.0
pspace (86)	**86**	**0.4**	73	17121.45	21	2444.97	80	12115.3	42	60.66	9	4435.97	62	12832.25	55	19321.93
rubik (7)	6	617.93	7	222.01	7	333.27	7	730.67	6	131.24	6	73.74	**7**	**153.49**	7	259.9
sage (26607)	26607	12652.48	26607	3345.83	**26607**	**1141.17**	26607	4999.07	26607	3367.8	26607	3747.98	26607	2912.67	26607	4432.5
spear (1695)	1692	10481.83	1689	4479.0	**1695**	**519.28**	1691	32950.37	1695	2828.45	1686	7530.68	1695	6299.7	1688	132297.38
stp (1)	1	84.63	1	22.93	**1**	**3.1**	1	12.23	1	15.72	1	20.78	1	16.0	0	0.0
stp_samples (426)	424	64.81	424	10.48	**424**	**2.34**	424	8.04	424	16.24	424	10.69	424	12.78	424	14.35
tacas07 (5)	5	127.65	5	110.0	5	997.83	5	261.56	5	582.51	5	443.46	5	99.2	**5**	**22.47**
uclid (416)	416	1235.1	416	307.4	**416**	**43.0**	416	773.83	416	153.92	416	425.43	416	213.02	416	480.3
uclid_contrib (7)	7	1061.88	7	188.35	7	2580.49	7	1505.94	7	1342.59	7	1188.97	**7**	**176.04**	7	212.71
uum (8)	2	26.28	**2**	**6.99**	2	28.07	2	8.79	2	17.58	2	8.46	2	9.65	2	36.44
wienand-cav2008 (18)	**14**	**15.09**	9	31.11	14	34.38	14	19.8	14	32.18	14	25.4	4	20.67	14	30.77
	31924	126778.97	31739	174232.78	31524	134438.95	**32046**	**169109.18**	31961	194727.1	31868	119845.65	31762	150716.02	31740	294646.69

Table 5.5: Comparing cvcLz with other solvers on QF_AUFBV.

	cvcLz		mathsat		z3		yices		boolector		sonolar	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
bench_ab (119)	119	6.76	119	5.23	119	1.51	**119**	**0.36**	119	2.74	119	1.99
bmc-arrays (39)	33	857.93	36	586.24	36	871.78	39	272.05	**39**	**126.51**	39	224.62
brummayerbiere (293)	240	10229.2	246	17998.08	226	12262.47	**255**	**10269.77**	250	13421.96	227	16965.29
brummayerbiere2 (22)	13	3745.89	14	543.68	12	1700.94	19	596.23	**21**	**583.73**	17	1550.37
brummayerbiere3 (10)	**10**	**1.08**	2	97.58	10	36.82	10	486.62	10	1.63	5	174.83
btfnt (1)	1	654.33	**1**	**12.4**	0	0.0	1	99.58	1	138.03	1	59.44
calc2 (36)	18	3671.36	33	5835.7	35	6828.49	31	3926.37	**36**	**824.59**	36	4348.85
dwp_formulas (5765)	5750	29415.72	5760	4517.19	5635	9354.35	5764	1905.49	**5765**	**3016.95**	5764	2558.45
ecc (92)	**91**	**123.06**	88	66.92	88	401.57	86	68.73	83	953.93	55	63.58
egt (7719)	7719	149.37	7719	155.67	7719	80.49	**7719**	**16.83**	7719	384.71	7719	47.69
jager (2)	0	0.0	0	0.0	0	0.0	**1**	**370.99**	0	0.0	0	0.0
klee-selected-smt2 (622)	622	8464.85	620	12170.37	**622**	**217.75**	622	1297.69	620	15093.03	622	6330.8
pipe (1)	1	307.91	0	0.0	1	21.45	1	22.56	1	10.83	**1**	**6.76**
platania (275)	236	3204.06	234	5436.06	241	1545.87	262	5474.78	**268**	**8653.39**	262	8583.37
sharing-is-caring (40)	**40**	**0.56**	40	40.21	40	1.07	40	62.51	40	2149.38	40	66.08
stp (40)	31	2431.17	29	1543.61	33	4234.48	**40**	**513.97**	38	1254.73	39	478.93
stp_samples (52)	52	4.33	52	1.52	52	1.45	**52**	**0.25**	52	1.12	52	1.09
	14976	63267.59	14993	49010.48	14869	37560.5	15061	25384.76	**15062**	**46617.23**	14998	41462.14

Finally, in Table 5.3, we compare cvcPll with other state-of-the-art bit-vector solvers (for a comparison of cvcVBS see Table 5.4). We chose the top performing solvers from the SMT solving competition SMTCOMP2014: yices, stp2, z3, boolector, 4simp, mathsat and sonolar. For all of the above solvers, we used the binaries and configurations entered in the SMT competition. For the parallel solver cvcPll, we report wall clock time.

The cvcPll solver solves significantly more problems than either cvcE or cvcLz and is competitive with the other solvers. The cvcPll solver solves the largest number of problems in the bruttomesso, calypto and pspace families due to the algebraic simplifications used during solving by cvcLz. We solve as many problems as the top solver on the brummayerbiere2 family due to the AIG simplifications in cvcE. Factoring out of isomorphic circuits helped us solve the most problems in the mcm family, when we initially ran the problems with a smaller timeout of 300 seconds. However, increasing the time limit allowed other solvers to catch up and even perform better. Although factoring out the circuits decreased the size of the bit-blasted formula for the entire family, it did not lead to a performance gain on harder problems. We speculate this may be related to the performance of the SAT solver we are using which is not as competitive as some of the SAT solvers used by the other SMT solvers.

We attribute the increased performance of cvcPll, compared to cvcE and cvcLz, to the complementary nature of the two approaches. Our solver cvcPll ranks 3^{rd} when compared with the other bit-vector solvers on all of the QF_BV SMT-LIB v2.0 2014-06-03 benchmarks. This is consistent with our standing on the QF_BV logic in the 2014 SMT-COMP after a minor bug-fix. A couple of comments on how the results reported here compare to those reported in [48]: in the results in [48], our solver cvcPll solves the most problems, while here we rank third. There are several reasons for this: (i) the paper experimental results were on a subset of QF_BV that did not include the asp family and the new log-slicing family, on which cvcPll performs worse than boolector and stp2; (ii) the different time and memory limits can affect which solver solves most problems in a certain family (as was the case for the mcm family).

Table 5.5 shows the performance of cvcLz on the QF_AUFBV logic. The table shows that cvcLz is competitive with other solvers. The performance on this family depends on many factors including the performance of the array theory solver as well as its interaction with the bit-vector solver. Next we will provide a more detailed analysis of the tradeoffs between the two approaches, based on our experimental results.

5.2 Lazy vs eager

We now provide a more detailed analysis of the tradeoffs between the two approaches, based on our experimental results.

The eager solver cvcE is particularly efficient on hardware equivalence checking benchmarks that verify the equivalence of a bit-level implementation to its word-level specification. In such cases, the correctness of the proof often depends on bit-level properties that benefit from efficient propositional analysis more than the kind of algebraic reasoning done in the lazy solver. This is especially obvious in the difference in the performance of cvcE and cvcLz on the brummayerbiere* families, as can be seen in Table 5.1.

Maintaining the word-level structure during the computation in LBV requires establishing a common language between the SAT solver driving the main CDCL($\mathcal{T}$) search (SAT$_{\mathsf{main}}$), and SAT$_{\mathsf{bb}}$. In our approach, this language consists of the $\mathcal{T}_{\mathsf{bv}}$-atoms and represents a frontier that partitions the problem between the two solvers. LBV conflicts can be seen as interpolants between the part of the problem describing the control flow (the Boolean abstraction) and the data-path. Restricting the conflict language to $\mathcal{T}_{\mathsf{bv}}$-atoms limits the granularity of the conflicts: we cannot express bit-level conflicts. In some cases this can prove inefficient. Consider the following example.

Example 17. The following assertions are unsatisfiable. All paths through the disjunction force the last bit of the x_i variables to be $0_{[1]}$. Therefore their disjunction must also have the least significant bit equal to $0_{[i]}$ which makes the equality false.

$$\bigvee_{i=0}^{n} x_i = y \circ 1_{[1]} \wedge \bigwedge_{i=0}^{n} (x_i = t_i \circ 0_{[1]} \vee x_i = s_i \circ 0_{[1]})$$

In Example 17, an eager solver may potentially learn that the last bit of each x_i has to be 0. The lazy solver on the other hand, will have to try all possible paths through the disjunction and learn a conflict for each one of them. Furthermore, this makes the lazy solver very sensitive to the encoding of the problem: the terms encoded as Booleans are sent to one SAT solver while the ones encoded as bit-vectors of size one, are sent to another. An unfortunate choice of which terms are Boolean can lead to poor performance, as illustrated by the float family in Table 5.1. The eager solver, however, does not distinguish between Boolean terms and bit-vectors of size one.

For problems with expensive arithmetic operators, the benefits of maintaining the word-level structure outweigh this limitation. While eager solvers have sophisticated rewrite techniques, such techniques are usually only applicable at the top level. Equivalence checking problems between higher level designs can require proving the equivalence of results obtained by taking different control-flow paths. These can be encoded as large ite (if-then-else) term trees with a similar structure, as in the following example.

Example 18. The formula below is unsatisfiable. The conditions on all paths through the ite trees force the leaves to be equal.

$$\begin{aligned} & ite(x_0 = y_0, x_0 * (ite(x_1 = y_1, 2 * x_1, 2)), 2) \neq \\ 2* \quad & ite(x_0 = y_0, y_0 * (ite(x_1 = y_1, y_1, 1)), 1) \end{aligned}$$

Collecting the assertions down any ite path in the example, and applying simple equality substitutions renders each such path trivially unsatisfiable. No multiplication reasoning is required. However, bitblasting this expression results in a difficult SAT problem as the large circuits required to model the products obscures the trivial inconsistency. The calypto family, and the bruttomesso sub-families lfsr and simple_processors (Table 5.1) exhibit this type of structure. On these families, our LBV in-processing module can often simplify each call to $\mathcal{T}$-check to false or a significantly simpler circuit. The lfsr family encodes the behavior of a linear feedback shift register: all full check queries can be reduced to *false* by our inprocessing xor solver, without requiring any SAT reasoning. The simple_processor family encodes a simple processor decoding instructions. The in-processing slicing allows for substituting the instruction codes and trivially reducing the problem to *false* without any SAT reasoning. Other verification problems, such as checking the correctness of sorting algorithms, rely on the arithmetic properties of a total order. Several sub-families of the platania family encode sorting algorithms in such a way that all assertions sent to the $\mathcal{T}_{\mathsf{bv}}$ solver consist only of equalities and inequalities. The $\mathsf{BV}_{\mathsf{eq}}$ and $\mathsf{BV}_{\mathsf{ineq}}$ sub-solvers are complete for these queries and no SAT reasoning is required. The bottleneck for these problems is the array axiom instantiations.

5.3 Related work

In this section, we discuss other related approaches to bit-vector solving, that did not fit in either the lazy or eager category. For related work on eager bit-vector solving see Section 3.5, and for lazy bit-vector solving, see Section 4.7.

Reduction to arithmetic. The main disadvantage of approaches based on bit-blasting is that they do not scale well with the width of the data-path particularly when arithmetic operations are involved. Modern SAT solvers are notoriously “bad at math”: they struggle to reason efficiently about large multipliers and division circuits.

To address this issue, some solvers encode the problem into a different domain such as integer arithmetic. Work by Zeng et al. [85] encodes bit-vector constraints into mixed integer linear programming (MILP) constraints. Arithmetic operations are modeled via arithmetic over bounded integers. Bitwise operations are linearized by introducing an integer variable $0 \leq t_i \leq 1$ for each bit of a term $t_{[n]}$, and asserting the following constraint:

$$t = \sum_{i=0}^{n-1} t_i \cdot 2^i.$$

The algorithm attempts to push as many of the constraints as possible in the arithmetic part and to avoid adding the bit-wise variables. This approach only supports linear bit-vector arithmetic. Work by Babic et al. [4] encodes bit-vector constraints using linear and non-linear modular arithmetic. For the non-linear case, their algorithm uses Newton’s p’adic iteration algorithm. Their algorithm is integrated into the Nelson-Oppen theory combination framework [69]. These solvers are efficient at dealing with large data-paths and arithmetic operations. However, linearization of shifts by a variable amount and bit-wise operations can lead to very large and challenging constraints.

Abstraction. A third class of approaches relies on an abstraction refinement loop [25, 58] which can significantly reduce the complexity of the problem. The work of [25] implemented in the UCLID [59] solver, alternates applying under and over-approximation and using them to refine each other. It first under-approximates the formula by restricting the number of Boolean variables used to represent bit-vector terms to the k least significant bits. The top bits are zero or sign-extended. If the under-approximation is satisfiable, so is the input formula. If it is unsatisfiable, the unsatisfiable core of the under-approximation is used to generate an over-approximation. The construction of the over-approximation is such that, if it is not precise enough, it can be used to infer how to refine the under-approximation. This technique assumes that the property to be checked does not depend on the exact functionality of the data-path. If this is not the case, it is unlikely that an appropriate abstraction will be found. The work of Gange et al. [45] applies abstraction to reasoning about bit-vector difference logic constraints. They adapt the Floyd-Warshall algorithm to give an incomplete decision procedure for bit-vector difference logic constraints. The algorithm does not have an abstraction refinement stage.

Rewriting. Word-level rewriting plays an important role in the performance of bit-vector solvers: most solvers implement hundreds of rewrite rules [42]. Due to the large number of operators as well as the counter-intuitive results of combining bit-level manipulations with arithmetic, coming up with the right set of rewrite rules is something of a black art. Recent work [67] attempts to tackle this challenge by automatically generating problem specific rewrite rules. It does so by adapting Stalmarck's algorithm for propositional logic. For each triplet of the form $t_1 \ op \ t_2 = t_3$ occurring in the input problem, a SAT solver is used to determine whether the triplet entails that any of the

t_1, t_2 or t_3 terms have a specific value. For example the triplet $x_{[8]} + (-x_{[8]}) = y_{[8]}$ entails that $y_{[8]} = 0_{[8]}$. This approach only learns bit-width specific rules. A similar idea was earlier explored in [50]. The author automatically generated bit-vector expressions and used a SAT solver to check which ones are equivalent. The rules were manually inspected to identify the width-independent ones, and implement them in STP2.

Chapter 6

Bit-vector Proofs

SMT solvers often decide problems ranging in complexity from NP-complete to undecidable. To achieve this, the solvers implement complex algorithms combining efficient SAT solving with theory-specific reasoning, requiring many lines of highly optimized code.[1] Because the solvers' code base changes frequently to keep up with the state of the art, bugs are still found in mature tools: during the 2014 SMT competition, five SMT solvers returned incorrect results. In a field where correctness is of paramount importance, this is particularly problematic. While great progress has been made in verifying complex software systems [55, 60], the verification of SAT and SMT solvers still remains a challenge [61].

One approach to addressing this concern is instrumenting the SMT solvers to emit a certificate of correctness. If the input problem is satisfiable, a natural certificate is a satisfying model (see Section 4.6). Correctness can be checked by evaluating the input formula using the model. In the unsatisfiable case, the solver could emit an externally-checkable proof of unsatisfiability. Proof checking algorithms and their implementa-

---

[1] For example the CVC4 code-base consists of over 250K lines of C++ code

tions usually consist of a small trusted core that implements a set of simple rules. These can be composed to prove complex goals, while maintaining trustworthiness.

Proof producing SMT solvers have been successfully used to improve the performance of interactive theorem provers, as shown in several recent papers [2, 13, 15, 18, 19, 46]. The interactive prover can discharge complex sub-goals to the SMT solver. It can then check or reconstruct the proof returned by the solver without having to trust the result. In some applications, such as interpolant generation [75] and certified compilation [26] the proof object itself is of interest, not just for establishing correctness.

This chapter presents a method of encoding and checking SMT-generated proofs for the bit-vector theory. Proof generation and checking for the bit-vector theory poses additional challenges compared to other theories. Algebraic reasoning is usually not sufficient by itself to decide most bit-vector formulas of practical interest. Reduction to SAT usually results in very large propositional proofs. In addition, the reduction itself must be proven correct. LFSC is a meta-logic that was specifically designed to serve as a unified proof format for SMT solvers. Encoding the $\mathcal{T}_{\mathsf{bv}}$ proof rules in LFSC helps address some of these challenges.

Section 6.1 provides a brief introduction to the LFSC proof language. It introduces its main features and motivates choosing it for encoding bit-vector proofs. This section also illustrates how to use LFSC to encode the kind of inferences routinely done by SMT solvers. The architecture of the proof generating module in the SMT solver CVC4 is presented in Section 6.2. Section 6.3 introduces the LFSC proof rules that are specific to the bit-vector theory. We conclude with related work in Section 6.4. This chapter assumes some familiarity with the basics of type theory.

6.1 LFSC

This section provides an introduction to the LFSC proof language. Previous work [79] shows how to use LFSC to encode and efficiently check the core constructs required by SMT generated proofs, such as CNF conversion and theory lemmas. We will briefly review these encodings in the rest of the section.

LFSC is an extension of the Edinburgh Logical Framework (LF) [51]. LF is a meta-framework: it allows for encoding custom defined proof systems as a collection of proof rules called a *signature*. This kind of flexibility is essential for SMT proofs, where each theory has its own theory-specific rules, and the algorithms used to decide them vary from solver to solver. However, while LF has been successfully used in a variety of applications such as encoding logics and modeling programming language semantics, pure LF is not well suited for encoding SMT generated proofs. LF-style declarative proof rules cannot always efficiently model the kind of high-powered reasoning usually employed by SMT solvers. To address this issue and efficiently check SMT-generated proofs, LFSC extends LF with computational side-conditions. These side-conditions consist of snippets of functional programming code that have to succeed, for the proof rule to be successfully applied. The side-condition code, along with the proof rule declarations become part of the trusted core and form the signature. The proof checker takes as input a signature containing all the proof rules, as well as a proof of unsatisfiability built using axioms in the signature. It then checks that the proof is correct w.r.t. the given signature.

In LF, proof rules are encoded as typing declarations, where the type represents the inference being made. For example, a declarative proof rule such as transitivity of inequality:

$$\frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3}\ \mathsf{ineq\_trans}$$

can be encoded in LF as a term of the following type:

$$\Pi t_1{:}\mathsf{term}.\, t_2{:}\mathsf{term}.\, t_3{:}\mathsf{term}.\, \Pi u_1{:}\mathsf{holds}\,(t_1 \leq t_2).\, \Pi u_2{:}\mathsf{holds}\,(t_2 \leq t_3).\, \mathsf{holds}\,(t_1 \leq t_3)\ .$$

The rule takes as arguments terms t_1, t_2 and t_3, as well as proofs of $t_1 \leq t_2$ and of $t_2 \leq t_3$, and it returns a proof of $t_1 \leq t_3$. Intuitively, the dependent type $\Pi\varphi{:}\mathsf{formula}.\mathsf{holds}(\varphi)$ represents a proof that the φ formula holds. Assuming previously declared type constructor `term` and the <= relation of type $\Pi t_1{:}\mathsf{term}.\, t_2{:}\mathsf{term}.\, \mathsf{term}$, the corresponding LFSC syntax is the following:

```
(declare ineq_trans (! t1 term
                    (! t2 term
                    (! t3 term
                    (! u1 (holds (<= t1 t2))
                    (! u2 (holds (<= t2 t3))
                          (holds (<= t1 t3)))))))
```

In LFSC syntax `!` represents the LF Π binder for the dependent function space. From now on, we will show most examples of proof rules in LFSC syntax and explain new syntax as it is introduced. For the full LFSC syntax and semantics see [78].

A common kind of inference made by SMT solvers consists of normalizing an expression by flattening it and combining like terms. This helps identify redundant structure. For example, most SMT solvers would prove the validity of the following arithmetic equality in one rewrite step:

$$(t_1 + (t_2 + (\ldots + t_n))) - (t_{i_1} + (t_{i_2} + (\ldots + t_{i_n}))) = 0$$

where $t_{i_1}, \ldots, t_{i_n}$ are a permutation of $t_1, \ldots, t_n$. However, encoding a proof of this equality in pure LF would require repeated applications of the associativity and commutativity rules, essentially amounting to sorting one of the two arguments of the $-$ until it is syntactically equal to the other.[2] Using LFSC's side condition code, the equality can be proven using a single rule application, as follows:

```
(declare eq_zero
       (! t term
       (^ (normalize t) 0)
            (holds (= t 0))))
```

where `normalize` is a function in the side-condition language that normalizes a linear equation. The `(^sc t)` syntax checks that the result returned by the side-condition `sc` matches the term `t`. In the above example this amounts to checking that `(normalize t)` is equal to `0`. If this is not the case or if the side-condition code throws an exception, the rule application will fail. The support for computational side-conditions will prove to be an essential feature when encoding bit-vector proofs in LFSC.

6.1.1 SMT proofs

SMT solvers combine SAT reasoning on the Boolean abstraction of the input formula with theory-specific solving. One can think of the $\mathcal{T}$-solvers as refining the propositional abstraction with $\mathcal{T}$-valid clauses until a contradiction can be derived purely on the propositional level. Section 1.5 shows how this interaction plays out by describing the CDCL($\mathcal{T}$) framework employed by most SMT solvers.

Most SAT solvers implement variations of the CDCL algorithm (Section 1.4). The *resolution* proof rule (see Section 1.4.2 for the definition) is refutationally complete for propositional logic [76] and has been successfully used as the basis for a common proof

---

[2] Example taken from [79].

format for SAT solvers [82]. SMT proofs, however, require several additional proof steps that are not necessary for SAT proofs: (i) the input formula for SMT solvers is rarely in CNF form, which means that a CNF conversion proof is necessary to establish that the clauses in the resolution proof follow from the input formula; (ii) the variables in the Boolean skeleton abstract $\mathcal{T}$-atoms, which means that an abstraction mechanism is required to make the connection between $\mathcal{T}$-atoms and the variables used to represent them in the SAT solver. Finally, each $\mathcal{T}$-valid fact must have a proof within theory $\mathcal{T}$ using $\mathcal{T}$-specific inferences. Since these facts are $\mathcal{T}$-valid, they can be proved without using any facts in the input formula.[3]

In this section, we will focus on how to encode these aspects of an SMT proof in the LFSC language. The SMT proof will have a two-tiered structure: a resolution tree whose leaves are either input clauses or $\mathcal{T}$-valid clauses; and the $\mathcal{T}$-valid clauses which are either generated as $\mathcal{T}$-conflicts, $\mathcal{T}$-lemmas or $\mathcal{T}$-explanation clauses of $\mathcal{T}$-propagations. The root of the resolution tree is the empty clause $\perp$.

6.1.2 Encoding Resolution

Before encoding the resolution proof rule, we first need a way to represent propositional clauses in LFSC. Propositional variables are represented by a var type and literals by a lit type. There are two type constructors for lit: pos and neg, both have type $\Pi x{:}\mathsf{var}.\mathsf{lit}$. They build a literal in which the variable occurs positively (pos) or negatively (neg) respectively. Clauses are modeled as lists of literals: the clause type also has two type constructors: cln and clc. The cln constructor corresponds to the empty clause $\perp$. The clc constructor has type $\Pi x{:}\mathsf{lit}.c{:}\mathsf{clause}.\mathsf{clause}$: intuitively it returns the clause obtained

---

[3] While SMT solvers can also reason about quantifiers and LFSC can express such proofs, we will focus on quantifier-free proofs in this chapter.

```
(declare var type)

(declare lit type)
(declare pos (! x var lit))
(declare neg (! x var lit))

(declare clause type)
(declare cln clause)
(declare clc (! x lit (! c clause clause)))
```

Figure 6.1: Encoding clauses in LFSC.

by appending literal x to clause c.

Figure 6.1 shows how to declare these types in LFSC syntax. For example, the clause $v_1 \vee \neg v_2 \vee v_3$ is encoded as the term:

```
  (clc (pos v_1) (clc (neg v_2) (clc v_3 cln)))
```

The dependent type $\Pi c{:}\mathsf{clause}.\mathsf{holds}(c)$ is used to denote proofs of clauses. Intuitively a term of the type $\mathsf{holds}(c)$ for some clause c, represents a proof that c holds. Using these constructs, we can now encode the resolution rule R. The proof rule takes as input clauses $c_1{:}\mathsf{clause}$, $c_2{:}\mathsf{clause}$ and $c_3{:}\mathsf{clause}$, as well as proofs that the first two clauses hold: $u_1{:}\mathsf{holds}(c_1)$ and $u_2{:}\mathsf{holds}(c_2)$. It also takes the variable $v{:}\mathsf{var}$ which will be used as the resolution pivot. The resolve side condition function defined elsewhere, computes the result of resolving clause c_1 with c_2. The side condition succeeds if resolving the two clauses on v was possible, and the resulting clause matches c_3. If this was the case the proof rule returns a proof of c_3: $\mathsf{holds}(c_3)$. Note that given c_1, c_2 and v the value of c_3 is fixed. For this reason, LFSC allows the use of holes _ when the value of a proof rule argument can be deduced from the others. To highlight arguments that can be left as a hole we will write them in the rule declaration in the following *`italicized`*

```
(declare holds (! c clause type))

(declare R (! c1 clause
           (! c2 clause
           (! c3 clause
           (! u1 (holds c1)
           (! u2 (holds c2)
           (! v var
           (^ (resolve c1 c2 v) c3)
              (holds c3))))))))
```

Figure 6.2: Encoding propositional resolution in LFSC.

`font`. Figure 6.2 shows the LFSC declaration of the resolution proof rule R.

Example 19. Consider proving the inconsistency of the following clauses:

$$\begin{array}{rcr} v_1 & \vee & \neg v_2 \\ v_1 & \vee & v_2 \\ \neg v_1 & & \end{array}$$

The following sequence of resolution steps proves the inconsistency by deriving the empty clause:

$$\dfrac{\dfrac{v_1 \vee \neg v_2 \quad v_1 \vee v_2}{v_1} \qquad \neg v_1}{\bot}$$

Figure 6.3 shows the corresponding proof in LFSC syntax. The $(\%$x t$)$ syntax represents $\lambda x{:}\tau.t$. The check command checks that the type annotation `(: type term)` is correct: term `term` has type `type`. In our example, the check command ensures that the clause computed by chaining the resolution rules has type holds(cln) (line 8).

For efficiency reasons, the actual LFSC signature we use lazily computes the resolved clause by marking the literals to be removed. This technique is called *deferred*

```
(check
(% v1 var
(% v2 var
(% v3 var
(% p1 (holds (clc (pos v1) (clc (neg v2) cln)))
(% p2 (holds (clc (pos v1) (clc (pos v2) cln)))
(% p3 (holds (clc (neg v1) cln))
(: (holds cln)
   (R _ _ _ (R _ _ _ p1 p2 v2) p3 v1)))))))))
```

Figure 6.3: Example resolution proof in LFSC.

resolution [72].

6.1.3 Encoding Theory Lemmas

Because the SAT solver in CDCL($\mathcal{T}$) reasons on the Boolean abstraction $\_^{\mathrm{P}}$ of the input formula, most SMT solvers must keep an internal mapping between the Boolean variables and the $\mathcal{T}$-atoms they abstract. In LFSC, this can be done using terms of type $\Pi v{:}\mathsf{var}.f{:}\mathsf{formula}.\mathsf{atom}(v, f)$. One can think of $\mathsf{atom}(v, f)$ as a predicate capturing the fact that variable v corresponds to formula f i.e. $f^{\mathrm{P}} = v$. The `decl_atom` proof rule provides a mechanism to introduce the atom constructs. It takes as arguments a formula f and a term u of type $\lambda v{:}\mathsf{var}.\lambda a : \mathsf{atom}(v, f).dtoholds\mathsf{cln}$. Intuitively, this term represents a proof of the empty clause, assuming there is a variable v that abstracts f. The `decl_atom` rule essentially says that if there is a way of deriving the empty clause by abstracting formula f with Boolean variable v, then you can derive the empty clause. We will show how `decl_atom` fits in the full LFSC proof in Example 22.

Given $\mathcal{T}$-valid lemma $l_1 \vee \ldots \vee l_n$, we want to build a proof of the corresponding $\mathrm{SAT}_{\mathsf{main}}$ clause $l_1^{\mathrm{P}} \vee \ldots \vee l_n^{\mathrm{P}}$. We proceed by assuming the negation of the $\mathcal{T}$-literals and deriving a contradiction. Using the fact that $\bigwedge \neg l_i \rightarrow \bot$ and the $\_^{\mathrm{P}}$ mapping (see

Section 1.5 for the definition of $\_^{\mathrm{P}}$) we will prove the Boolean abstraction clause.

The `assume_true` and `assume_false` rules will be used to introduce the negation of the $\mathcal{T}$-literals. Recall that we introduced the dependent type $\Pi c{:}\mathsf{clause}.\mathsf{holds}(c)$ to represent a proof of the clause c. Similarly, we introduce the following dependent type to denote proofs of formulas:

$$\Pi f{:}\mathsf{formula}.\mathsf{th\_holds}(f)$$

The `assume_true` proof rule takes a $\mathcal{T}$-formula f as well as the SAT variable v used to abstract it (this is encoded by a term of type $\mathsf{atom}(v, f)$). It additionally takes as an argument u, a function that builds a propositional proof of clause c ($\mathsf{holds}(c)$) assuming a proof of formula f ($\mathsf{th\_holds}(f)$). Intuitively u states $f \Rightarrow c$. This is equivalent to $\neg f \vee c$. Since v was the variable representing f, this is the same as the conclusion of the `assume_true` rule: a proof of the clause $\neg v \vee c$.

For example, say we want to prove the $\mathcal{T}$-valid lemma $x = x$ represented in the SAT solver as the unit clause v_0, i.e. $v_0 = (x = x)^{\mathrm{P}}$. The term $a_0{:}\mathsf{atom}(v_0, x = x)$ encodes this fact and will be one of our assumptions. We will use the `assume_false` rule. The proof rule needs as an argument some term that derives the empty clause assuming the literal: $l_0{:}\mathsf{th\_holds}(\neg(x = x))$. Let us assume l_0 holds and show how to use this fact to derive the empty clause. A term of type $\mathsf{th\_holds}(x = x)$ can be built using the equality symmetry rule ($\mathsf{eq\_sym}\ x$):

```
(declare eq_sym (! t term
                  (th_holds (= t t))))
```

Using the assumption l_0, and the contradiction proof rule ($\mathsf{contra}\ (\mathsf{eq\_sym}\ \mathsf{x})\ \mathsf{l}_0$) we can build a term of the type $\mathsf{th\_holds}(\mathit{false})$, where contra is defined as follows:

```
(declare th_holds (! formula type))
(declare atom (! v var (! f formula type)))

(declare decl_atom
           (! f formula
           (! u (! v var
                (! a (atom v f)
                     (holds cln)))
           (holds cln))))

(declare clausify_false
  (! u (th_holds false)
       (holds cln)))

(declare assume_true
  (! v var
  (! f formula
  (! c clause
  (! r (atom v f)
  (! u (! o (th_holds f)
            (holds c))
    (holds (clc (neg v) c))))))))

(declare assume_false
  (! v var
  (! f formula
  (! x clause
  (! r (atom v f)
  (! u (! o (th_holds (not f))
            (holds c))
    (holds (clc (pos v) c))))))))
```

Figure 6.4: Encoding the mapping between $\mathcal{T}$-atoms and their Boolean abstraction in LFSC.

```
(declare contra
      (! f formula
      (! u1 (th_holds f)
      (! u2 (th_holds (not f))
            (th_holds false)))))
```

However, building a proof of *false* signifies an inconsistency and is equivalent to deriving the empty clause as shown by the `clausify_false` rule in Figure 6.4. Using this rule, we can build the term: $\lambda o{:}\mathsf{th\_holds}(\neg x = x).\mathsf{clausify\_false}(\mathsf{contra}\ (\mathsf{eq\_sym}\ x)\ o)$ with type $\Pi o{:}\mathsf{th\_holds}(\neg x = x).\mathsf{holds}(\mathsf{cln})$. This is exactly the type of the final argument required by `assume_false`. Putting it all together using LFSC syntax the following term has type $\mathsf{holds}(\mathsf{clc}\ v_0\ (\mathsf{clc}\ \mathsf{cln}))$:

```
(assume_false _ _ _ a0 (\ l0
        (clausify_false (contra _ (eq_sym x)))))
```

where the `\ l0` syntax represents the λ binding $\lambda l_0.t$. Recall that a_0 was a term of type $\mathsf{atom}(v_0, x = x)$.

The same kind of reasoning can be used to prove a more complex $\mathcal{T}$-valid clause of the form $l_1 \vee \ldots \vee l_n$. Chaining applications of `assume_true` and `assume_false` builds the following implication:

$$\neg l_1 \Rightarrow \neg l_2 \ldots \Rightarrow \neg l_n \Rightarrow \bot$$

Example 20. We will show how to prove the following $\mathcal{T}_{\mathsf{uf}}$-valid clause using LFSC proof rules:

$$\neg x = y \vee \neg y = z \vee x = z.$$

Figure 6.5 shows the LFSC proof that shows the validity of the lemma on the Boolean abstraction. The comments indicate the type of the intermediate proof terms. The equality atoms $x = y, y = z, x = z$ correspond to the SAT variables v_1, v_2 and v_3 of type var. On line 6, the literal l_1 is a proof of $x = y$ and has type th_holds($x = y$). Similarly for l_2 and l_3. We build a proof of the fact that $x = z$ by applying the equality transitivity proof rule trans (line 13) defined as follows:

```
(declare trans
    (! t1 term
    (! t2 term
    (! t3 term
    (! u1 (th_holds (= t1 t2))
    (! u2 (th_holds (= t2 t3))
          (th_holds (= t1 t3))))))))
```

A contradiction is then derived contradiction using the contra proof rule.

The clausify_false rule turns a proof of the *false* formula (th_holds(*false*)) into a proof of the empty clause (holds(cln)). Since we proved a contradiction from the negation of the literals, the chained assume_true and assume_false proof rules build the final clause on the Boolean abstraction of the $\mathcal{T}$-literals. Therefore the check command succeeds as the computed type of the chained assume_$*$ rules represents a proof of the desired clause.

6.1.4 CNF Conversion

Different CNF conversion algorithms require different proof rules. In this section we will focus on Tseitin-style CNF encodings, as they are usually used by many SMT solvers including CVC4. CNF conversion proofs can be built in a similar fashion as $\mathcal{T}$-lemmas. Instead of using $\mathcal{T}$-specific proof rules to derive a contradition from the negation of the clause, we can use propositional natural deduction proof rules to derive a contradition.

```
(check
(% a1 (atom v1 (= x y))
(% a2 (atom v2 (= y z))
(% a3 (atom v3 (= x z))
(: (holds (clc (neg v3) (clc (neg v2) (clc (pos v3) cln))))
(assume_true _ _ _ a1 (\ l1
;; (holds (clc (neg v2) (clc (pos v3) cln)))
(assume_true _ _ _ a2 (\ l2
;; (holds (clc (pos v3) cln))
(assume_false _ _ _ a3 (\ l3
(clausify_false                ;; (holds cln)
   (contra _                   ;; (th_holds false)
      (trans _ _ _ _ l1 l2)    ;; (th_holds (= x z))
      l3)))))))))))))))))      ;; (th_holds (not (= x z)))
```

Figure 6.5: Theory lemma example proof.

```
(declare or_elim
  (! f1 formula
  (! f2 formula
  (! u1 (th_holds (not f2))
  (! u2 (th_holds (or f1 f2))
    (th_holds f1))))))

(declare impl_elim
  (! f1 formula
  (! f2 formula
  (! u1 (th_holds f1)
  (! u2 (th_holds (impl f1 f2))
    (th_holds f2))))))
```

$$\frac{\varphi_1 \vee \varphi_2 \quad \neg\varphi_2}{\varphi_1}\ \textsf{or\_elim}$$

$$\frac{\varphi_1 \Rightarrow \varphi_2 \quad \varphi_1}{\varphi_2}\ \textsf{impl\_elim}$$

Figure 6.6: Example proof rules for CNF encoding proofs in LFSC.

Example 21. Using the propositional logic natural deduction rules from Figure 6.6, we can encode a proof of the CNF conversion of the following formula:[4]

$$(a \Rightarrow b) \vee c \quad \overset{\text{CNF}}{\Longrightarrow} \quad \neg a \vee b \vee c$$

Figure 6.7 shows the CNF conversion proof in LFSC syntax. We will show that the clause is entailed by the input formula, by assuming the entailment doesn't hold and deriving a contradiction:

$$((a \Rightarrow b) \vee c) \wedge (a \wedge \neg b \wedge \neg c).$$

As before we use the `assume_true` and `assume_false` rules to introduce the negation of the literals in the clause as assumptions in the proof. The or_elim rule uses $\neg c$ to simplify $(a \Rightarrow b) \vee c$ to $a \Rightarrow b$. Since we assumed a, we can use the impl_elim rule to simplify the implication to derive b. However, we also assumed $\neg b$, hence we can derive a contradiction using contra followed by clausify_false. The cascading assume_$*$ proof rules complete the proof.

Tseitin-style CNF conversion can introduce fresh propositional variables for intermediate formulas. The above approach can handle this case, since the atom dependent type constructor can bind an arbitrary formula to a SAT variable, not just a $\mathcal{T}$-atom. For example if the CNF conversion introduces intermediate propositional variable v for sub-formula $a \wedge b$, we can use atom(v, $a \wedge b$) to prove the clauses corresponding to $v \Leftrightarrow (a \wedge b)$.

---

[4] A traditional Tseitin encoding would have added an intermediate variable for the implication. For brevity we do not do that in this example.

```
(check
(% F  (th_holds (or (impl a b) c))
(% a1 (atom v1 a)
(% a2 (atom v2 b)
(% a3 (atom v3 c)
( : (holds (clc (pos v1) (clc (neg v2) (clc (neg v3) cln))))
(assume_true _ _ _ a1 (\ l1
(assume_false _ _ _ a2 (\ l2
(assume_false _ _ _ a3 (\ l3
(clausify_false
       (contra _                     ;; (th_holds false)
            (impl_elim _ _ l1        ;; (th_holds b)
                (or_elim _ _ l3 F))  ;; (th_holds (impl a b))
        l2)))))))))
```

Figure 6.7: LFSC CNF conversion proof example.

6.2 Proofs in CVC4

This section first presents the architecture of the proof-generating module in the SMT solver CVC4 and then introduces the $\mathcal{T}_{\mathsf{bv}}$ LFSC proof signature. Proof generation can add significant overhead to solving. In such cases, we wanted CVC4's proof module to be as unintrusive as possible and add zero overhead when proof production is disabled. Several design decisions were made to achieve this goal:

- Proof generation is restricted to a ProofManager object. Other parts of the system log just enough proof hints to the ProofManager so that it can reconstruct the final proof.

- For most theories $\mathcal{T}$-conflicts are replayed in proof-producing $\mathcal{T}$-solvers. This means that most $\mathcal{T}$-solvers do not need to log any proofs during the initial run.

- Proof logging code is compiled out if proofs are not required.

The ProofManager has several modules, as shown in Figure 6.8. During CNF conversion, the CNF mapping between $\mathcal{T}$-formulas and their corresponding SAT variables as well as the steps taken during CNF conversion are stored in the CNF proof. The CDCL solver $\text{SAT}_{\mathsf{main}}$ is instrumented to store resolution steps of the inferences it makes such as learning new clauses during conflict resolution (see Section 1.4). For each clause learned by $\text{SAT}_{\mathsf{main}}$, the SatProof module stores a resolution chain which starts from the falsified conflict clause and derives the learned clause. Note that although some of these clauses are deleted when the SAT solver clause database is cleared, their resolution chains must be kept. Clauses learned from the deleted clauses may still exist in the clause database.

The TheoryProof must keep track of the learned $\mathcal{T}$-lemmas and which $\mathcal{T}$-solver generated them (we use the term $\mathcal{T}$-lemmas to refer generically to $\mathcal{T}$-conflicts, $\mathcal{T}$-lemmas and $\mathcal{T}$-explanation clauses). The proof module corresponding to the theory that generated the lemma will be responsible for providing a proof of the lemma. We assume all $\mathcal{T}$-lemmas are in clausal form.

The SatProof builds the full unsatisfiability proof by working backwards from the final conflict, and stitching together the already stored resolution chains. The leaves of the resolution tree are either input clauses (which have a CNF conversion proof from the input formula), or $\mathcal{T}$-lemmas. Because not all the $\mathcal{T}$-lemmas will appear in the resolution tree, for most theories we generate the $\mathcal{T}$-lemma proofs lazily. Since every $\mathcal{T}$-lemma is valid, a new instance of the $\mathcal{T}$-solver with proof production enabled is in principle able to return a $\mathcal{T}$-proof. This general approach works well for $\mathcal{T}_{\mathsf{uf}}$ and $\mathcal{T}_{\mathsf{arr}}$ proofs. For the bit-vector theory, however, we record the proofs for the conflicts eagerly. Re-proving $\mathcal{T}_{\mathsf{bv}}$ conflicts will likely require calls to a SAT solver which are potentially very expensive.

Finally, the ProofManager uses the proofs built by the individual modules to construct a proof that derives the empty clause, starting from the input formula. The CnfProof establishes that all the input clauses are entailed from the input formula, the TheoryProof module creates a proof for each $\mathcal{T}$-lemma and the SatProof builds the final resolution tree from these proven clauses.

Example 22. We will now show a full example of an LFSC SMT generated proof for the following unsatisfiable set of assertions:

$$x = 0 \vee y = 0$$

$$x * y \neq 0$$

where x and y are integer variables. An SMT solver could prove the unsatisfiability of the formula by first learning the following two theory lemmas:

$$x \neq 0 \vee x * y = 0$$

$$y \neq 0 \vee x * y = 0$$

and then letting the SAT solver derive a contradiction by using purely propositional reasoning. In LFSC the proof can be encoded as follows:

```
(check
(% x (term Int)
(% y (term Int)
(% A1 (th_holds (or (= Int x (toInt 0)) (= Int y (toInt 0))))
(% A2 (th_holds (not (= Int (mult x y) (toInt 0))))
(: (holds cln)
(decl_atom (= Int x (toInt 0)) (\ v1 (\ a1
(decl_atom (= Int y (toInt 0)) (\ v2 (\ a2
(decl_atom (= Int (mult x y) (toInt 0)) (\ v3 (\ a3
;; CNF conversion proof of clause [v1,v2]
(@c1 (assume_false _ _ _ a1 (\ l1
     (assume_false _ _ _ a2 (\ l2
```

```
        (clausify_false (contra _ (or_elim _ _ l2 A1)
                              l1))))))
;; CNF conversion proof of clause [~v3]
(@c2 (assume_true _ _ _ a3 (\ l3
        (clausify_false (contra _ l3 A2))))
;; Proof of lemma x != 0 or x * y = 0
;; represented by clause [~v1, v3]
(@lem1 (assume_true _ _ _ a1 (\ l1
       (assume_false _ _ _ a3 (\ l3
          (clausify_false (contra _ (mult_zero1 x y l1) l3))))))
;; Proof of lemma y != 0 or x * y = 0
;; represented by clause [~v1, v3]
(@lem2 (assume_true _ _ _ a2 (\ l2
       (assume_false _ _ _ a3 (\ l3
          (clausify_false (contra _ (mult_zero2 x y l2) l3))))))
;; Resolution proof
(R _ _  (R _ _  c1 (R _ _  lem1 c2 v3) v1)
        (R _ _  lem2 c2 v3) v2)
)))))))))))))))))))
```

In the above the `A1` and `A2` formulas represent the input assertions from which the proof of unsatisfiability will be derived. The `Int` type is a simple type used to denote integers, and `toInt` builds a constant integer. The overall computed type of the final proof term will be:

$$\lambda x{:}\mathsf{term}(\mathsf{Int}).\lambda y{:}\mathsf{term}(\mathsf{Int}).$$

$$\lambda A_1{:}\mathsf{th\_holds}(x = 0 \lor y = 0).$$

$$\lambda A_2{:}\mathsf{th\_holds}(x * y \neq 0).\mathsf{holds}(\mathsf{cln})$$

The `decl_atom` proof rule defined in Figure 6.4 introduces the propositional variables v_1, v_2 and v_3 abstracting $\mathcal{T}$-atoms $x = 0$, $y = 0$ and $x * y = 0$ respectively. It also introduces the terms a_1, a_2 and a_3 of type `atom`(v, f) that establish the connection between the propositional variables and the atoms. The `(@l t f)` syntax stands for the let-binding "let l be t in $f[l]$". The `mult_zero1` proof rule in the lemma proofs is used to show that if the first multiplicand is zero the entire product is also zero. It has the following declaration:

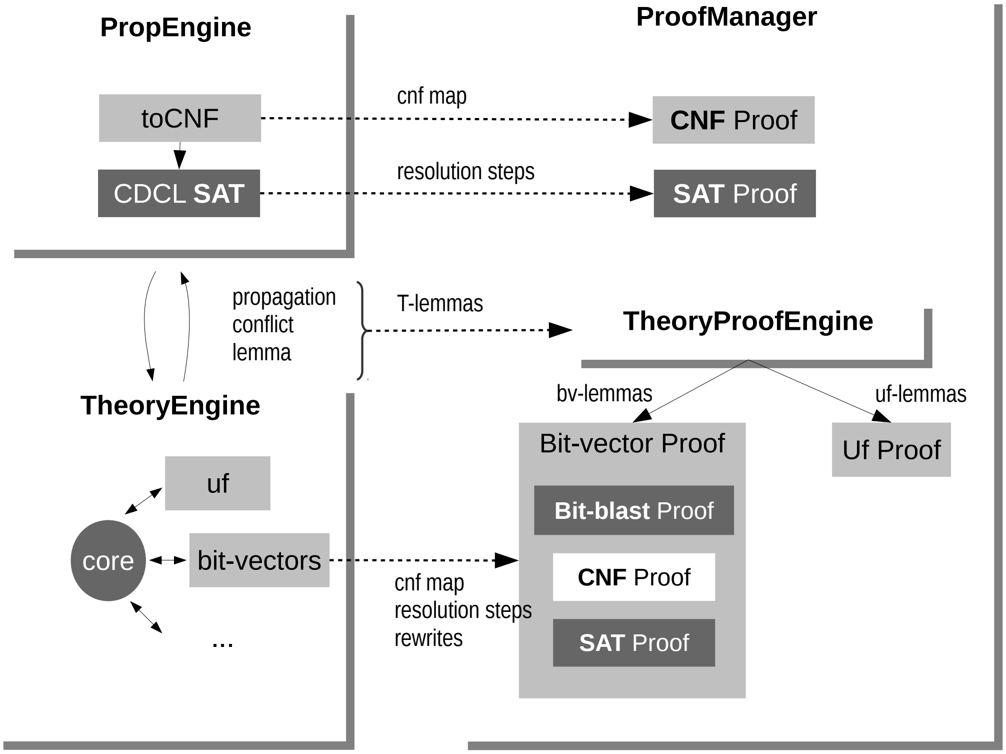

Figure 6.8: CVC4 proof module architecture.

```
(declare mult_zero1
         (! x (term Int)
         (! y (term Int)
         (! u (th_holds (= Int x 0))
              (th_holds (= Int (mult x y) 0))))))
```

The `mult_zero2` rule is identical, except it takes a proof of the second multiplicand by zero.

6.3 Bit-vector Proofs

This section introduces an LFSC proof signature for encoding $\mathcal{T}_{\mathsf{bv}}$ proofs generated by the SMT solver CVC4. Proof generation for the bit-vector theory in CVC4 targets the lazy bit-vector solver cvcLz described in Chapter 4. Eager bit-vector solvers eagerly reduce the problem to SAT so most of the reasoning is done in a SAT solver. A proof generated by such a solver would be dominated by a (potentially very large) propositional resolution proof and lose any structure present in the input problem. The lazy approach allows for a modular proof: the resolution proof in $\text{SAT}_{\mathsf{main}}$ is separated from the proofs of the individual $\mathcal{T}_{\mathsf{bv}}$-conflicts. The $\mathcal{T}_{\mathsf{bv}}$-conflicts proved by the algebraic subsolvers can have word-level algebraic proofs. Note that the $\mathsf{BV}_{\mathsf{eq}}$ and $\mathsf{BV}_{\mathsf{ineq}}$ proofs can be expressed using simple natural deduction rules such as congruence and transitivity of equality and inequality. Most of these rules are bit-width independent, and do not require reasoning about the bit-blasted terms. The $\mathsf{BV}_{\mathsf{bb}}$ conflicts, still require their own resolution based proof and a proof that the bit-blasting is correct.

Bit-blasting is one of the most challenging aspects of representing $\mathcal{T}_{\mathsf{bv}}$ proofs using declarative rules. The features of LFSC, particularly the side-condition language, help facilitate this encoding. For the rest of this section, we will focus on the LFSC signature required to encode and check $\mathsf{BV}_{\mathsf{bb}}$ conflicts.

Figure 6.9 shows the overall structure of a $\mathcal{T}_{\mathsf{bv}}$ proof. The resolution proof generated by $\text{SAT}_{\mathsf{main}}$ proves unsatisfiability by deriving the empty clause. The leaves of the resolution tree are either input clauses, obtained by converting to CNF the input problem, or $\mathcal{T}_{\mathsf{bv}}$-lemmas. Each $\mathcal{T}_{\mathsf{bv}}$-lemma has a corresponding resolution proof in SAT_{bb}. Note however that the SAT variable a^{BB} abstracting $\mathcal{T}_{\mathsf{bv}}$-atom a in the resolution proof in SAT_{bb} is not necessarily the same as the a^{P} variable used to abstract the same atom in $\text{SAT}_{\mathsf{main}}$. Therefore, we need to maintain this mapping explicitly. The leaves of the SAT_{bb} resolution tree must all be definitional bit-blasting clauses C^{BB}, obtained from CNF conversion of the atom definitions. The bit-blasting proof introduces the atom definition by constructing a proof of the fact that each atom a is equivalent to its bit-blasted formula: $a \Leftrightarrow \mathsf{bbAtom}(a)$. This proof requires no assumptions as $a \Leftrightarrow \mathsf{bbAtom}(a)$ is

$\mathcal{T}_{\mathsf{bv}}$-valid.[5]

6.3.1 Bit-vector theory LFSC signature

Our encoding of SMT formulas in LFSC distinguishes between formulas and terms. The formula type is a simple type, while the term type is a dependent type parametrized by the sort of the term: $\Pi s{:}\mathsf{sort}.\mathsf{term}(s)$. The sort type is also a simple type and it is used to distinguish between terms of different types. The sort $[n]$ of all bit-vectors of length n, is represented in LFSC as the dependent type: $\Pi n{:}\mathsf{Int}.\mathsf{BitVec}(n)$. Figure 6.10 shows the concrete LFSC declarations of these constructs. The `mpz` type is LFSC's own built-in infinite precision integer type.

Bit-vector constants are represented as a sequence of bits using the const_bv type with the two type constructors bvn and bvc. The const_bv bit-vector constants are converted to bit-vector terms using the toBV rule in Figure 6.10. The side condition code calls the bv_len procedure define elsewhere, that returns the length of a term of type const_bv. This ensures that the bit-width of the constant bit-vector is the same as that of the term to be returned.

Using these constructs, most bit-vector operators can be declared in a very straightforward way. For example the $\&$ bit-wise and operator can be encoded in LFSC as follows:

```
(declare bvand
   (! n mpz
   (! x (term (BitVec n))
   (! y (term (BitVec n))
      (term (BitVec n))))))
```

---

[5] Technically, to express this in the $\mathcal{T}_{\mathsf{bv}}$ signature we would need to use a 1-bit extract $v[i:i]$ for the i^{th} bits of a bit-vector variable v.

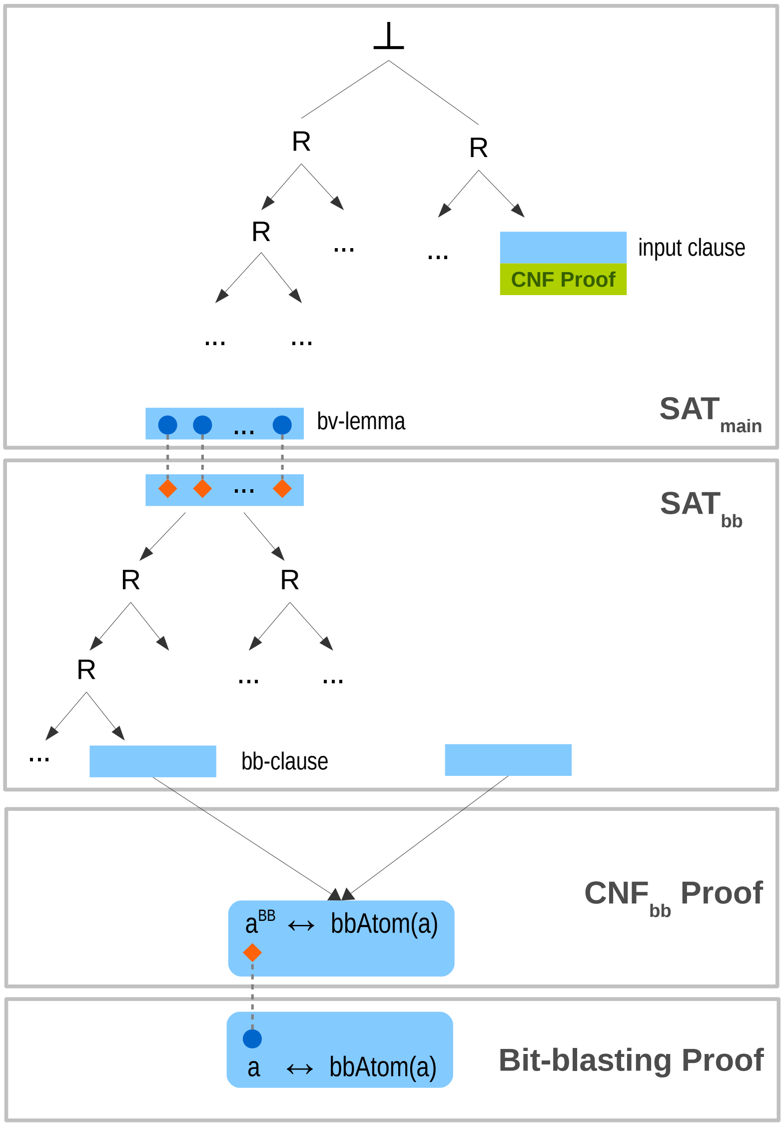

Figure 6.9: Bit-vector resolution proof diagram.

```
(declare formula type)
(declare sort type)
(declare term (! t sort type))

(declare BitVec (!n mpz sort))

(declare b0 bit)
(declare b1 bit)

(declare const_bv type)
(declare bvn const_bv)
(declare bvc (! b bit (! v const_bv const_bv)))

(declare toBV
 (! n mpz
 (! v const_bv
     (^ (bv_len v) n)
    (term (BitVec n)))))
```

Figure 6.10: Bit-vector theory LFSC signature.

The argument and result types ensure that it is applied to arguments of the right bit-width. Because the value of the bit-width n can be inferred from the x and y arguments, it can be left as an LFSC hole: `(bvand _ x y)`.

Operators that return bit-vectors of bit-widths different than that of the arguments can be encoded using side-condition code as follows:

```
(declare concat
   (! n mpz
   (! m mpz
   (! m' mpz
   (! t1 (term (BitVec n))
   (! t2 (term (BitVec m))
   (^ (mp_add n m) m')
      (term (BitVec m')))))))))
```

The value of `m'` does not need to be specified: the side-condition code uses the built-in addition function `mp_add` to ensure that the length of the returned bit-vector is the sum of the lengths of the two arguments.

Example 23. The $\mathcal{T}_{\mathsf{bv}}$ formula $(t_1 \circ t_1 = t_2 + t_3) \vee (t_1 < 0_{[3]})$ can be encoded in LFSC concrete syntax as:

```
(or (= (BitVec 6) (concat _ _ _ t1 t1) (bvadd _ t2 t3))
    (bvult _ t1 (toBV 3 (bvc b0 (bvc b0 (bvc b0 bvn))))))
```

6.3.2 Resolution in $\mathrm{SAT}_{\mathsf{bb}}$

The bit-blasting sub-solver $\mathsf{BV}_{\mathsf{bb}}$ relies on the solve-with-assumptions infrastructure described in Section 1.4.3 to check the satisfiability of the conjunction of $\mathcal{T}_{\mathsf{bv}}$-literals A_{bv}. Recall that all the learned clauses in CDCL-style SAT solvers are entailed by the initial problem clauses. Algorithm 3 in Section 1.4 shows how the learned clause is derived by several resolution steps from the already existing clauses. For each learned clause $\mathsf{learned} = [l_1, \ldots, l_n]$ we store a resolution chain of the form:

$$\mathsf{Res}(\ldots \mathsf{Res}(\mathsf{Res}(\mathsf{conflict}, \mathsf{reason}(l_{i_1})), \mathsf{reason}(l_{i_2})), \mathsf{reason}(l_{i_k}))$$

where the sequence of literals $l_{i_1} \ldots l_{i_k}$ corresponds to the order in which literals are processed in the loop in Algorithm 3 and $\mathsf{conflict}$ is the falsified conflict clause. The structure of the resolution proof follows directly from the way the $\mathsf{learned}$ clause is computed. Recall that assumption conflicts in $\mathsf{BV}_{\mathsf{bb}}$ are also derived by resolving out existing clauses (Algorithm 9 in Section 4.2) and are entailed by the problem clauses. The $\mathsf{SatProof}$ module in $\mathrm{SAT}_{\mathsf{bb}}$ stores these individual resolution chains during solving. This allows us to be able to reconstruct the resolution proof of all $\mathcal{T}_{\mathsf{bv}}$-conflicts returned by $\mathsf{BV}_{\mathsf{bb}}$, starting from the bit-blasting clauses C^{BB}.

As the clauses learned by $\mathrm{SAT}_{\mathsf{bb}}$ are kept between $\mathcal{T}_{\mathsf{bv}}$-check calls and could be involved in future conflicts, it is important to be able to reuse the resolution proofs of learned clauses for multiple $\mathcal{T}_{\mathsf{bv}}$ conflicts, without having to reprove them each time.

6.3.3 Bit-blasting

At the bottom of the proof in Figure 6.9 is the bit-blasting proof. The bit-blasting proof makes the connection between the bit-vector formula and its propositional logic encoding by proving for each bit-blasted atom a in the input formula, the following $\mathcal{T}$-valid formula:

$$a \Leftrightarrow \mathsf{bbAtom}(a).$$

We represent a bit-blasted bit-vector term of width n as a sequence of n formulas with the i^{th} formula corresponding to the i^{th} bit. The bblt type encodes bit-blasted terms and has two type constructors bbltn and bbltc as shown in Figure 6.11. We introduce the dependent type constructor bblast_term to encode the fact that a bit-blasted term y:bblt corresponds to the bit-vector term x:BitVec(n). This is conceptually similar to the atom construct introduced earlier. For example, the following term represents a proof that $\mathsf{bbTerm}(15_{[4]}) = [true, true, true, true]$:

```
(bblast_term _
  (toBV 4 (bvc b1 (bvc b1 (bvc b1 (bvc b1 bvn)))))
          (bbltc true (bbltc true (bbltc true (bbltc true bbltn))↩
             )))
```

Using the bit-blasting terms bblt we can define proof rules that build up more complicated bit-blasted terms. For example, the proof rule that establishes how to bit-blast the bit-vector and operator is the following:

```
  (declare bv_bbl_bvand
```

```
(declare bblt type)
(declare bbltn bblt)
(declare bbltc (! f formula (! v bblt bblt)))

(declare bblast_term
     (! n mpz
     (! x (term (BitVec n))
     (! y bblt
        type))))
```

Figure 6.11: Bit-blasting proof signature in LFSC.

```
       (! n mpz
       (! x (term (BitVec n))
       (! y (term (BitVec n))
       (! xb bblt
       (! yb bblt
       (! rb bblt
       (! xbb (bblast_term n x xb)
       (! ybb (bblast_term n y yb)
       (^ (bblast_bvand xb yb ) rb)
            (bblast_term n (bvand n x y) rb)))))))))))
```

The rule takes a proof that x_b is the bit-blasted term corresponding to x, and y_b corresponding to y, and it returns a proof that $x\&y$ is bit-blasted to r_b. The r_b bit-blasting term is constructed by the side-condition code bblast_bvand (see Appendix for the side-condition code of bblast_bvand and other more involved operators).

Bit-blasting $\mathcal{T}_{\mathsf{bv}}$-atoms follows a similar pattern:

```
(declare bv_bbl_eq
       (! n mpz
       (! x (term (BitVec n))
       (! y (term (BitVec n))
       (! bx bblt
       (! by bblt
       (! f formula
       (! bbx (bblast_term n x bx)
       (! bby (bblast_term n y by)
```

```
          (^ (bblast_eq bx by) f)
             (th_holds (iff (= (BitVec n) x y) f))))))))))))
```

Note that the bit-blasting proof rule does not take any $\mathcal{T}_{\mathsf{bv}}$-assertions as assumptions. Its conclusion is $\mathcal{T}_{\mathsf{bv}}$-valid.

Example 24. Encoding in LFSC the bit-blasting proof for the following formula $a_{[8]} = x_{[8]} \& y_{[8]}$ requires the following proof rule applications:

```
(bv_bbl_eq _ _ _ _ _ _ _ _ _
   (bv_bbl_var _ a)
   (bv_bbl_and _ _ _ _ _ _ (bv_bbl_var _ x) (bv_bbl_var _ y)))
```

The type of the above term is $\mathsf{holds}(\varphi)$ for φ:

$$(a_{[8]} = x_{[8]} \& y_{[8]}) \Leftrightarrow \left(\bigwedge_{0 \leq i < 8} a_i \Leftrightarrow x_i \wedge y_i \right).$$

where a_i is the i^{th} bit of bit-vector variable a.

The bit-blasting definition formulas can now be converted to CNF using the method described in Section 6.1.4. This allows us to prove all the learned clauses in SAT$_{\mathsf{bb}}$ and that all the assumption conflicts are entailed by the bit-blasted clauses C^{BB}.

Recall from the proof diagram in Figure 6.9 that there is a disconnect between the SAT variables in SAT$_{\mathsf{bb}}$ and those in SAT$_{\mathsf{main}}$. We will now show how to bridge this gap. Say SAT$_{\mathsf{bb}}$ identifies a conflict in the current set of $\mathcal{T}_{\mathsf{bv}}$-assertions A_{bv}. It computes the inconsistent assumption literals by resolving backwards from the assumption conflict (Algorithm 10 in Section 4.2). We can build a resolution chain from SAT$_{\mathsf{bb}}$ using the bit-blasted clauses C^{BB} and prove the clause:

$$c^{\mathsf{BB}} : [l_1^{\mathsf{BB}}, \ldots, l_n^{\mathsf{BB}}]$$

```
(declare intro_assump_true
     (! f formula
     (! v var
     (! C clause
     (! h (th_holds f)
     (! a (atom v f)
     (! u (! unit (holds (clc (pos v) cln))
             (holds C))
     (holds C))))))))
```

Figure 6.12: Encoding $\mathsf{BV_{bb}}$ conflicts in LFSC.

However, the clause used in the resolution in $\mathrm{SAT_{main}}$ is:

$$c^{\mathrm{P}} : [l_1^{\mathrm{P}}, \ldots, l_n^{\mathrm{P}}]$$

Recall that the assume_true and assume_false proof rules allowed us to transform a proof of $\bigwedge_{i=0}^{n} \neg l_i \Rightarrow \bot$ to a proof of the clause $[l_1^{\mathrm{P}}, \ldots, l_n^{\mathrm{P}}]$. We will use similar rules intro_assump_true and intro_assump_false (Figure 6.12) to prove the $\mathrm{SAT_{bb}}$ clause c^{BB} along with the negation of the $\mathcal{T}_{\mathsf{bv}}$-literals $\bigwedge_{i=0}^{n} \neg l_i$ entail $\bot$. The intro_assump_$*$ rules introduce the negation of the $\mathcal{T}_{\mathsf{bv}}$ literal as a unit clause, that can be resolved with c^{BB} to derive the empty clause.

Example 25. We show how to put these rules together to lift a proof of a clause in $\mathrm{SAT_{bb}}$ to a proof of the corresponding clause in $\mathrm{SAT_{main}}$. In the LFSC expression below, assume c is a proof of the $\mathrm{SAT_{bb}}$ clause $[\neg a_1^{\mathsf{BB}}, a_2^{\mathsf{BB}}]$, that at_1, at_2, b_1 and b_2 are atoms declared elsewhere and have types at_1:$\mathsf{atom}(a_1^{\mathrm{P}}, a_1)$, at_2:$\mathsf{atom}(a_2^{\mathrm{P}}, a_2)$, b_1:$\mathsf{atom}(a_1^{\mathsf{BB}}, a_1)$ and b_2:$\mathsf{atom}(a_2^{\mathsf{BB}}, a_2)$ respectively. Due to the declaration of the assume_$*$ rules, the l_1 and l_2 arguments to intro_assump_$*$ must have types $\mathsf{th\_holds}(f_1)$ and $\mathsf{th\_holds}(\neg f_1)$ respectively. The two resolution steps between the assumption unit clauses $unit_1$ and

$unit_2$ derive the empty clause from c, which allows the assume_$*$ proof rules to prove the $\mathcal{T}$-lemma $[\neg a_1, a_2]$.

```
(assume_true _ _ _ at1 (\ l1
(assume_false _ _ _ at2 (\ l2
   (intro_assump_true _ _ _ l1 b1 (\unit1
   (intro_assump_false _ _ _ l2 b2 (\unit2
          (R _ _ (R _ _ c unit1 v1) unit2 v2)))))))))
```

6.3.4 Rewriting

Most SMT solver rewriting systems, CVC4 included, combine several simpler rewrite rules that are applied to a fix-point. To allow for easily generating rewrite proofs in this manner, we introduce the LFSC dependent type constructors: rw_term(t_1, t_2) and rw_formula(f_1, f_2) that encode the fact that term t_1 rewrites to term t_2, and formula f_1 rewrites to formula f_2 respectively. The rw_term and rw_formula types are meant to record intermediate rewrites steps. They do not have the idempotence property mentioned in Section 2.1.

Using these types, we can chain intermediate rewrite steps together to prove more complex rewrite rules. If a subterm s of term t rewrites to s', so does the term $t[s \leftarrow s']$. To capture this kind of substitution reasoning we define proof rules for term constructors that "build" up the rewritten term. This kind of procedure closely mimics what SMT solvers usually do: expressions are often represented using an immutable data-structure, so changing a sub-expression would require rebuilding the entire expression.

Example 26. We will now show how to combine simpler rewrite rules to prove the following rewrite:

$$\neg((\sim\sim x_{[32]}) = x_{[32]}) \overset{\text{Rewriter}}{\Longrightarrow} \bot.$$

```
;; (th_holds false)
(apply_rw _ _ phi
  ;; (rw_formula (not (= (bvnot (bvnot x)) x)) false)
  (rw_trans _ _ _
    ;; (rw_formula (not true) false)
    not_true
    ;; (rw_formula (not (= (bvnot (bvnot x)) x)) (not true))
    (rw_not _ _
      ;; (rw_formula (= (bvnot (bvnot x)) x) true)
      (eq_rw _ _ _
      (rw_bvnot _ _ _   ;; (rw_term (bvnot (bvnot x)) x)
         (rw_id _ x)))) ;; (rw_term x x)
```

The commented lines show the types of the intermediate expressions. We begin by using the identity rewrite to prove that x rewrites to itself rw_term(x, x), in order to be able to establish the assumption for the rw_bvnot proof rule. The eq_rw proof asserts that if a term t_1 rewrites to another term t_2, the $t_1 = t_2$ equality must hold. The rw_not rule says that if a term t rewrites to t' then $\neg t$ also rewrites to $\neg t'$. Rules like this one allow for simulating substitution, by rebuilding terms that have a rewritten sub-term. The not_true rule rewrites $\neg\top$ to $\bot$, and the rw_trans rule takes advantage of a fact that if t_1 rewrites to t_2 and t_2 to t_3, then t_1 rewrites to t_3. Finally, the apply_rw rule asserts that if formula ϕ holds, and we have proven that ϕ rewrites to ϕ', then ϕ' must also hold.

6.4 Related Work

The work in [65] shows how to use an external checker to efficiently check proofs generated by the Fx7 solver for the AUFLIA SMT-LIB v2.0 logic. Several other approaches have relied on using interactive theorem provers to certify proofs produced by SMT solvers. The work in [46] shows how to use HOL Light to certify proofs generated by the SMT solver CVC3 for the AUFLIA SMT-LIB v2.0 logic. In [41], proofs for quantifier-free problems in the logic of equality with uninterpreted symbols are gen-

```

(declare rw_term
     (! s sort
     (! t (term s)
     (! t' (term s)
     type))))

(declare rw_formula
     (! f formula
     (! f' formula
     type)))

(declare rw_not
     (! a formula
     (! a’ formula
     (! rw (rw_formula _ a a')
        (rw_term _ (not _ a) (not _ a'))))))

(declare bvnot_idemp
     (! n mpz
     (! a (term (BitVec n))
     (! a’ (term (BitVec n))
     (! rwa (rw_term _  a a')
        (rw_term _ (bvnot _ (bvnot _ a)) a'))))))

(declare eq_rw
     (! n mpz
     (! a (term (BitVec n))
     (! a’ (term (BitVec n))
     (! rwa (rw_term _  a a')
        (rw_formula (= (BitVec n) a a') true))))))

(declare apply_rw_formula
     (! f formula
     (! f’ formula
     (! rw (rw_formula f f')
     (! fh (th_holds f)
         (th_holds f'))))))
```

Figure 6.13: Rewrite proofs in LFSC.

erated by the haRVey SMT solver and translated into Isabelle/HOL to provide more automation. While using an interactive theorem prover to check the proofs generated by SMT solvers has the advantage of ensuring a high level of trust, long proof-checking times make this approach difficult to scale. The work in [18] seeks to bridge this performance gap by optimizing the proof reconstruction step to reduce proof-checking time. However this work does not apply to bit-vector theory proofs.

The work in [47] also targets SMT generated proofs for the theory of bit-vectors for interpolant generation. This work also builds on a lazy bit-vector solver integrated in the DPLL($\mathcal{T}$) framework, to exploit the word-level structure for producing word-level interpolants. If the algebraic solvers fail, the fallback is the resolution proof generated by the SAT solver. Because this work specifically targets generating interpolants, it does not cover proving the correctness of bit-blasting or of rewrite rules.

The work in [17] shows how to do proof reconstruction in Isabelle/HOL for the theory of bit-vectors based Z3's proofs. However, as the authors remark, the coarse granularity of Z3's bit-vector proofs made proof reconstruction particularly challenging. By contrast, our approach is very fine-grained as it provides a full resolution proof for each bit-vector conflict.

The LFSC meta-framework has been successfully used for encoding proofs generated by SMT solvers in [72, 74]. The work in [75] shows how to use LFSC to compute interpolants from unsatisfiability proofs in the theory of equality and uninterpreted function symbols. We believe that using the approach in [75], it is possible to extend our proof system to generate bit-vector interpolants from LFSC bit-vector proofs.

Chapter 7

Conclusion

The performance of formal verification tools is tightly coupled with that of their back-end theorem proving engines. Verifying many safety critical systems require reasoning about fixed-width bit-vectors. This thesis investigated new techniques of solving bit-vector constraints, and evaluated their performance. The eager bit-vector solver presented in Chapter 3 leverages AIG simplification techniques and SAT reasoning to efficiently solve bit-vector constraints that require low level of bit reasoning. The technique for factoring out isomorphic circuits explores redundancy in the input formula and allows for reducing the size of the bit-blasted problem.

The lazy bit-vector solver presented in Chapter 4, maintains the word-level structure during solving and leverages this information to solve problems that are usually challenging for an eager solver. This solver also explores how to efficiently combine a bit-blasting sub-solver and integrate it in the CDCL($\mathcal{T}$) framework by providing a tight coupling with the main SAT engine. These techniques proved complementary to the eager approach. Combining the lazy and eager approaches in a parallel solver results in a dramatic improvement in performance, as shown in Chapter 5.

Finally, in Chapter 6 we showed how to increase the trustworthiness of the SMT solver by instrumenting it with proof generating capabilities. We also presented a proof system for machine checkable SMT generated proofs for the bit-vector theory. These proofs can exploit the word-level structure maintained by the lazy solver to reduce the proof size.

Appendix A

LFSC bit-blasting signature code

```
;; A predicate to represent the n^th bit of a bitvector term
(declare bitof
     (! x var_bv
     (! n mpz formula)))

;; A bit-vector variable
(declare var_bv type)

;; A bv variable term
(declare a_var_bv
     (! n mpz
     (! v var_bv
        (term (BitVec n)))))

;; Calculate the length of a bit-blasted term
(program bblt_len ((v bblt)) mpz
  (match v
    (bbltn 0)
    ((bbltc b v') (mp_add (bblt_len v') 1))))

;;Introduce the declaration of a bit-blasted term
(declare decl_bblast
     (! n mpz
     (! b bblt
     (! t (term (BitVec n))
     (! bb (bblast_term n t b)
```

```
     (! s (^ (bblt_len b) n)
     (! u (! v (bblast_term n t b) (holds cln))
           (holds cln))))))))

;; Bit-blast a variable
(program bblast_var ((x var_bv) (n mpz)) bblt
  (mp_ifneg n bbltn
              (bbltc (bitof x n) (bblast_var x (mp_add n (~ 1))))))

(declare bv_bbl_var (! n mpz
                    (! x var_bv
                    (! f bblt
                    (! c (^ (bblast_var x (mp_add n (~ 1))) f)
                         (bblast_term n (a_var_bv n x) f))))))

;; Bit-blast bvand
(program bblast_bvand ((x bblt) (y bblt)) bblt
  (match x
    (bbltn (match y (bbltn bbltn) (default (fail bblt))))
    ((bbltc bx x') (match y
                      (bbltn (fail bblt))
                      ((bbltc by y') (bbltc (and bx by)
                                            (bblast_bvand x' y'))
                                            )))))


(declare bv_bbl_bvand
    (! n mpz
    (! x (term (BitVec n))
    (! y (term (BitVec n))
    (! xb bblt
    (! yb bblt
    (! rb bblt
    (! xbb (bblast_term n x xb)
    (! ybb (bblast_term n y yb)
    (! c (^ (bblast_bvand xb yb ) rb)
       (bblast_term n (bvand n x y) rb)))))))))))

;; Bit-blast bvadd
(program bblt_bvadd_carry
        ((a bblt) (b bblt) (c formula)) formula
(match a
  ( bbltn (match b (bbltn c) (default (fail formula))))
```

```
  ((bbltc ai a') (match b
                  (bbltn (fail formula))
                  ((bbltc bi b') (or (and ai bi)
                                     (and (xor ai bi)
                                          (bblt_bvadd_carry a' b'↩
                                              c)
                                      )))))))

;; Ripple carry adder where carry is the initial carry bit
(program bblast_bvadd ((a bblt) (b bblt) (carry formula)) bblt
(match a
  ( bbltn (match b (bbltn bbltn) (default (fail bblt))))
  ((bbltc ai a') (match b
                   (bbltn (fail bblt))
                   ((bbltc bi b') (bbltc (xor ai (xor bi (↩
                      bblt_bvadd_carry a' b' carry)))
                     (bblast_bvadd a' b' carry)))))))

(declare bv_bbl_bvadd
   (! n mpz
   (! x (term (BitVec n))
   (! y (term (BitVec n))
   (! xb bblt
   (! yb bblt
   (! rb bblt
   (! xbb (bblast_term n x xb)
   (! ybb (bblast_term n y yb)
   (! c (^ (bblast_bvadd xb yb false) rb)
        (bblast_term n (bvadd n x y) rb)))))))))))
```

Bibliography

[1] ACKERMANN, W. *Solvable cases of the decision problem.* North-Holland, 1962.

[2] ARMAND, M., FAURE, G., GRÉGOIRE, B., KELLER, C., THÉRY, L., AND WERNER, B. A modular integration of SAT/SMT solvers to coq through proof witnesses. In *Certified Programs and Proofs*. 2011.

[3] ARMANDO, A., MANTOVANI, J., AND PLATANIA., L. Bounded model checking of software using SMT solvers instead of SAT solvers. In *SPIN Workshop on Model Checking of Software* (2006).

[4] BABIĆ, D., AND MUSUVATHI, M. Modular arithmetic decision procedure. *Microsoft Research Redmond, Tech. Rep. TR-2005-114* (2005).

[5] BARNETT, M., YUH EVAN CHANG, B., DELINE, R., JACOBS, B., AND LEINO, K. R. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects* (2006).

[6] BARRETT, C., CONWAY, C., DETERS, M., HADAREAN, L., JOVANOVIĆ, D., KING, T., REYNOLDS, A., AND TINELLI, C. CVC4. In *Computer Aided Verification* (2011).

[7] BARRETT, C., AND DONHAM, J. Combining SAT methods with non-clausal decision heuristics. *Electronic Notes in Theoretical Computer Science* (2005).

[8] BARRETT, C., NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. Splitting on demand in SAT modulo theories. In *Logic for Programming, Artificial Intelligence, and Reasoning* (2006).

[9] BARRETT, C., STUMP, A., AND TINELLI, C. The smt-lib standard: Version 2.0. In *Workshop on Satisfiability Modulo Theories* (2010).

[10] BARRETT, C., AND TINELLI, C. Satisfiability modulo theories. In *Handbook of Model Checking*. 2014. In preparation.

[11] BARRETT, C. W., DILL, D. L., AND LEVITT, J. R. A decision procedure for bit-vector arithmetic. In *Design Automation Conference* (1998).

[12] BAYLESS, S., VAL, C. G., BALL, T., HOOS, H. H., AND HU, A. J. Efficient modular SAT solving for IC3. In *Formal Methods in Computer-Aided Design* (2013).

[13] BESSON, F., CORNILLEAU, P.-E., AND PICHARDIE, D. Modular SMT proofs for fast reflexive checking inside Coq. In *Certified Programs and Proofs*. 2011.

[14] BJESSE, P., AND BORALV, A. Dag-aware circuit compression for formal verification. In *Computer-aided Design* (2004).

[15] BLANCHETTE, J. C., BÖHME, S., AND PAULSON, L. C. Extending Sledgehammer with SMT solvers. *Journal of automated reasoning* (2013).

[16] BOBOT, F., FILLIÂTRE, J.-C., MARCHÉ, C., AND PASKEVICH, A. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages* (2011).

[17] BÖHME, S., FOX, A., SEWELL, T., AND WEBER, T. Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL. In *Certified Programs and Proofs*. 2011.

[18] BÖHME, S., AND WEBER, T. Fast LCF-style proof reconstruction for Z3. In *Interactive Theorem Proving* (2010).

[19] BOUTON, T., CAMINHA B. DE OLIVEIRA, D., DÉHARBE, D., AND FONTAINE, P. veriT: An open, trustable and efficient SMT-solver. In *Conference on Automated Deduction* (2009).

[20] BRAYTON, R., AND MISHCHENKO, A. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification* (2010).

[21] BRUMMAYER, R., AND BIERE, A. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2009.

[22] BRUTTOMESSO, R. *RTL Verification: from SAT to SMT (BV)*. PhD thesis, PhD thesis, University of Trento, 2008. Available at http://www. inf. unisi. ch/post-doc/bruttomesso/files/phdthesis. pdf, 2008.

[23] BRUTTOMESSO, R., CIMATTI, A., FRANZEN, A., GRIGGIO, A., HANNA, Z., NADEL, A., PALTI, A., AND SEBASTIANI, R. A lazy and layered SMT BV solver for hard industrial verification problems. In *Computer Aided Verification* (2007).

[24] Bruttomesso, R., and Sharygina, N. A scalable decision procedure for fixed-width bit-vectors. In *International Conference on Computer Aided Design* (2009).

[25] Bryant, R. E., Kroening, D., Ouaknine, J., Seshia, S. A., Strichman, O., and Brady, B. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2007.

[26] Chen, J., Chugh, R., and Swamy, N. Type-preserving compilation of end-to-end verification of security enforcement. In *Programming Language Design and Implementation* (2010).

[27] Cimatti, A., Griggio, A., Schaafsma, B. J., and Sebastiani, R. The MathSAT5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2013.

[28] Clarke, E. M., Biere, A., Raimi, R., and Zhu, Y. Bounded model checking using satisfiability solving. *Formal Methods in System Design* (2001).

[29] Cook, S. A. The complexity of theorem-proving procedures. In *Symposium on Theory of Computing* (1971).

[30] Cyrluk, D., Möller, O., and Ruess, H. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Computer Aided Verification* (1997).

[31] de Moura, L., and Bjørner, N. Relevancy propagation. Tech. Rep. MSR-TR-2007-140, Microsoft Research, 2007.

[32] De Moura, L., and Bjørner, N. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[33] DETLEFS, D., NELSON, G., AND SAXE, J. Simplify: a theorem prover for program checking. *Journal of the ACM* (2005).

[34] DUTERTRE, B., AND DE MOURA, L. The yices SMT solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf* (2006).

[35] EÉN, N., AND BIERE, A. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing* (2005).

[36] EÉN, N., AND SÖRENSSON, N. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science* (2003).

[37] EÉN, N., AND SÖRENSSON, N. An extensible SAT-solver. In *Theory and applications of satisfiability testing* (2004).

[38] ENDERTON, H. B. *A mathematical introduction to logic*. 2001.

[39] EZICK, J., SPRINGER, J., HENRETTY, T., AND OH, C. Extreme SAT-based Constraint Solving with R-Solve. In *IEEE Conference on High Performance Extreme Computing* (2014).

[40] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., AND SAXE, J. B. Extended static checking for Java. In *Programming Language Design and Implementation* (2002).

[41] FONTAINE, P., MARION, J. Y., MERZ, S., NIETO, L. P., AND TIU, A. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *In Tools and Algorithms for Construction and Analysis of Systems* (2006).

[42] FRANZÉN, A. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, 2010.

[43] FUJIWARA, H., MEMBER, S., SHIMONO, T., AND MEMBER, S. On the acceleration of test generation algorithms. *IEEE Transactions on Computers* (1983).

[44] GANESH, V., AND DILL, D. L. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification* (2007).

[45] GANGE, G., SØNDERGAARD, H., STUCKEY, P. J., AND SCHACHTE, P. Solving difference constraints over modular arithmetic. In *Conference on Automated Deduction* (2013).

[46] GE, Y., AND BARRETT, C. Proof translation and SMT-LIB benchmark certification: A preliminary report. In *Workshop on Satisfiability Modulo Theories* (2008).

[47] GRIGGIO, A. Effective word-level interpolation for software verification. In *Formal Methods in Computer-Aided Design* (2011).

[48] HADAREAN, L., BANSAL, K., JOVANOVIC, D., BARRETT, C., AND TINELLI, C. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *Conference on Computer Aided Verification* (2014).

[49] HAGEN, G., AND TINELLI, C. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *Formal Methods in Computer-Aided Design* (2008).

[50] HANSEN, T. *A Constraint Solver and its Application to Machine Code Test Generation*. PhD thesis, University of Melbourne, 2012.

[51] HARPER, R., HONSELL, F., AND PLOTKIN, G. A Framework for Defining Logics. *Journal of the Association for Computing Machinery* (1993).

[52] JOVANOVIĆ, D., AND BARRETT, C. Sharing is caring: combination of theories. In *Frontiers of combining systems*. 2011.

[53] JUNKER, U. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *Association for the Advancement of Artificial Intelligence* (2004).

[54] KIM, H., SOMENZI, F., AND JIN, H. Efficient term-ite conversion for satisfiability modulo theories. In *Theory and Applications of Satisfiability Testing-SAT 2009*. 2009.

[55] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Symposium on Operating Systems Principles* (2009).

[56] KOTHARI, N., MAHAJAN, R., MILLSTEIN, T. D., GOVINDAN, R., AND MUSUVATHI, M. Finding protocol manipulation attacks. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2011).

[57] KOVÁSZNAI, G., FRÖHLICH, A., AND BIERE, A. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *Workshop on Satisfiability Modulo Theories* (2012).

[58] KROENING, D., AND SESHIA, S. A. Formal verification at higher levels of abstraction. In *Computer-aided design* (2007).

[59] LAHIRI, S. K., AND SESHIA, S. A. The uclid decision procedure. In *Computer Aided Verification* (2004).

[60] LEROY, X. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages* (2006).

[61] LESCUYER, S., AND CONCHON, S. A Reflexive Formalization of a SAT Solver in Coq. In *Theorem Proving in Higher Order Logics* (2008).

[62] LOPES, N. P., AKSOY, L., MANQUINHO, V., AND MONTEIRO, J. Optimally solving the MCM problem using pseudo-boolean satisfiability. *arXiv preprint arXiv:1011.2685* (2010).

[63] MANZANO, M. Introduction to many-sorted logic. In *Many-sorted Logic and its Applications* (1993).

[64] MISHCHENKO, A., CHATTERJEE, S., AND BRAYTON, R. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In *Design Automation Conference* (2006).

[65] MOSKAL, M. Rocket-Fast Proof Checking for SMT Solvers. In *Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[66] NADEL, A. Handling bit-propagating operations in bit-vector reasoning. In *Workshop on Satisfiability Modulo Theories* (2013).

[67] NADEL, A. Bit-vector rewriting with automatic rule generation. In *Computer-aided design* (2014).

[68] NADEL, A., RYVCHIN, V., AND STRICHMAN, O. Ultimately incremental SAT. In *Theory and Applications of Satisfiability Testing*. 2014.

[69] NELSON, G., AND OPPEN, D. C. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems* (1979).

[70] NIEMETZ, A., PREINER, M., AND BIERE, A. Turbo-charging lemmas on demand with donâĂŹt care reasoning. *Formal Methods in Computer-Aided Design* (2014).

[71] NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. Solving SAT and SAT modulo theories: From an abstract DPLL procedure to DPLL(T). *Journal of the ACM* (2006).

[72] OE, D., REYNOLDS, A., AND STUMP, A. Fast and Flexible Proof Checking for SMT. In *Workshop on Satisfiability Modulo Theories* (2009).

[73] PREINER, M., NIEMETZ, A., AND BIERE, A. Lemmas on demand for lambdas. *Program Proceedings* (2013).

[74] REYNOLDS, A., HADAREAN, L., TINELLI, C., GE, Y., STUMP, A., AND BARRETT, C. Comparing proof systems for linear real arithmetic with LFSC. In *Workshop on Satisfiability Modulo Theories* (2010).

[75] REYNOLDS, A., TINELLI, C., AND HADAREAN, L. Certified interpolant generation for EUF. In *Workshop on Satisfiability Modulo Theories* (2011).

[76] ROBINSON, J. A. *Logic: Form and Function: The Mechanization of Deductive Reasoning*. 1979.

[77] SÖRENSSON, N., AND EÉN, N. Minisat 2.1 and Minisat++ 1.0 - SAT race. *Theory and Applications of Satisfiability Testing* (2009).

[78] STUMP, A. Proof checking technology for satisfiability modulo theories. *Electronic Notes in Theoretical Computer Science* (2009).

[79] STUMP, A., OE, D., REYNOLDS, A., HADAREAN, L., AND TINELLI, C. SMT proof checking using a logical framework. *Formal Methods in System Design* (2013).

[80] STUMP, A., SUTCLIFFE, G., AND TINELLI, C. Starexec: a cross-community infrastructure for logic solving. In *International Joint Conference on Automated Reasoning* (2014).

[81] TSEITIN, G. S. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*. 1983.

[82] VAN GELDER, A. http://users.soe.ucsc.edu/ avg/ProofChecker/ProofChecker-fileformat.txt.

[83] WARREN, H. S. *Hacker's delight*. 2013.

[84] ZEE, K., KUNCAK, V., AND RINARD, M. C. An integrated proof language for imperative programs. In *Programming Language Design and Implementation* (2009).

[85] ZENG, Z., KALLA, P., AND CIESIELSKI, M. LPSAT: A unified approach to RTL satisfiability. In *Conference on Design, Automation and Test in Europe* (2001).