JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Dipl.-Ing.
Aina Niemetz, Bsc

Angefertigt am
Institut für Formale
Modelle und Verifikation

Erstbeurteiler
Univ.-Prof. Dr.
Armin Biere

Zweitbeurteiler
Assoc. Prof.
Clark Barrett

Februar 2017

Bit-Precise Reasoning Beyond Bit-Blasting

Dissertation

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

im Doktoratsstudium der

Technischen Wissenschaften

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

In the field of hardware and software verification, many applications require to determine satisfiability of first-order-logic with respect to one or more background theories, also referred to as Satisfiability Modulo Theories (SMT). The majority of these applications relies on bit-precise reasoning as provided by SMT solvers for the quantifier-free theory of fixed-size bit-vectors, often combined with arrays and uninterpreted functions. Fixed-size bit-vectors provide a natural way to model circuits and programs and arrays allow to reason about memory and array data structures. Uninterpreted functions, on the other hand, are useful as abstraction for irrelevant or too complex details of a system.

In this thesis, our main focus is on SMT procedures for bit-vector logics. In the context of quantifier-free bit-vector formulas in SMT, current state-of-the-art is a flattening technique referred to as bit-blasting, where the input formula is eagerly translated into propositional logic and handed to an underlying SAT solver. While efficient in practice, in particular for increasing bit-widths, bit-blasting may not scale if the input size can not be reduced sufficiently during preprocessing. In this thesis, we propose alternative approaches for bit-vector logics based on local search that do not require bit-blasting or an underlying SAT solver and yield a substantial gain in performance, in particular in combination with bit-blasting within a sequential portfolio setting.

In the context of combining quantifier-free bit-vectors with arrays and uninterpreted functions, current state-of-the-art SMT procedures are based on lazy rather than eager techniques. One such lazy technique is the Lemmas on Demand (LOD) approach, which refines full candidate models of a formula abstraction with lemmas until convergence. Full candidate models, however, include irrelevant parts of the input formula, which may introduce unnecessary overhead. In this thesis, we propose an optimization of LOD where focusing refinement on relevant parts of the input formula considerably improves performance.

We implemented all of our techniques within our SMT solver Boolector, which contributed to Boolector winning several tracks of recent SMT competitions. Boolector supports the quantifier-free theories of fixed-size bit-vectors, arrays and uninterpreted functions and natively handles non-recursive first-order lambda terms. It is a complex piece of software with correctness, robustness and high performance as its key requirements. and in this thesis, we address automated testing and debugging techniques for SMT solver development that we consider as crucial to reach this goal.

Zusammenfassung

Eine Vielzahl an Applikationen im Bereich der formalen Verifikation von Hardware und Software erfordert das Lösen des Erfüllbarkeitsproblem der Prädikatenlogik erster Stufe unter Berücksichtigung einer oder mehrerer Theorien, auch bekannt als Satisfiability Modulo Theories (SMT), und setzt dafür auf SMT-Prozeduren (sogenannte SMT-Solver) für die quantorenfreien Theorien der Bitvektoren, Arrays und nicht-interpretierten Funktionen.

Die bisher gebräuchliste und in der Praxis effizienteste Technik für das Lösen von quantorenfreien Bitvektorformeln ist "Bit-Blasting", das eine gegebene Formel in ein Entscheidungsproblem der Aussagenlogik (SAT) übersetzt und an einen SAT-Solver delegiert. Für hohe Bitweiten kann es jedoch sein, dass Bit-Blasting nicht skaliert wenn es dem Präprozessor nicht gelingt die gegebene Formel ausreichend zu vereinfachen bevor sie nach SAT übersetzt wird. Wir stellen eine Methode vor, Bitvektorformeln in SMT ohne Bit-Blasting zu lösen, die auf lokaler Suche basiert, und insbesondere in Kombination mit Bit-Blasting in einem so genannten sequentiellen Portolio eine erhebliche Leistungsverbesserung erzielt.

Aktuell gebräuchliche Prozeduren für das Lösen von Bitvektorformeln, die Arrays und nicht-interpretierte Funktionen enthalten, sind im Gegensatz zu Bit-Blasting keine direkten Übersetzungen nach SAT, sondern basieren auf Ansätzen, die eine Abstraktion der gegebenen Formel iterativ verfeinern. Ein Beispiel dafür ist die "Lemmas on Demand" (LOD) Technik, die vollständige Modelle der Formelabstraktion mit Lemmas verfeinert bis Erfüllbarkeit oder Nichterfüllbarkeit nachgewiesen werden kann. Vollständige Modelle enthalten jedoch auch irrelevante Teile der Formelabstraktion, was unnötigen Overhead erzeugen kann. Wir stellen eine Optimierung von LOD vor, die nur relevante Teile der Formelabstraktion berücksichtigt und damit die Leistung deutlich verbessert.

Wir haben alle unsere Techniken in unserem SMT-Solver Boolector implementiert, was erheblich dazu beigetragen hat, dass Boolector mehrere Tracks der SMT Competitions der letzten Jahre gewonnen hat. Boolector ist ein SMT-Solver für die quantorenfreien Theorien der Bitvektoren, Arrays und nicht-interpretierten Funktionen und bietet darüberhinhaus native Unterstützung für nichtrekursive Lambdaterme erster Ordnung. Boolector ist ein komplexes Tool, das in erster Linie als Backend eingebunden wird, und als Hauptanforderungen Korrektheit, Robustheit und hohe Performanz erfüllen muss. Wir stellen automatisierte Test- und Debugging-Techniken für die SMT-Solverentwicklung vor, die wesentlich dazu beigetragen haben diese Ziele zu erreichen.

Contents

I Prologue **1**

1 Introduction **3**
1.1 Outline and Contributions 4

2 Background **7**
2.1 SAT 7
2.2 SMT 8
2.3 Bit-Blasting in Boolector 11

3 Paper A: Improving Local Search for Bit-Vector Logics in SMT with Path Propagation **13**
3.1 Discussion 13

4 Paper B: Propagation Based Local Search for Bit-Precise Reasoning **21**
4.1 Discussion 21

5 Paper C: Turbo-Charging Lemmas on Demand with Don't Care Reasoning **25**
5.1 Discussion 26

6 Paper D: ddSMT: A Delta Debugger for the SMT-LIB v2 Format **31**
6.1 Discussion 32

7 BtorMBT: A Model-Based API Tester for Boolector **33**
7.1 Workflow 34
7.2 Test Case Generation with BtorMBT 35
7.3 API Trace Execution with BtorUntrace 39
7.4 API Error Trace Minimization with ddMBT 41
7.5 Experimental Evaluation 41
7.6 Discussion 45

8 Conclusion **47**

II Papers **49**

A Improving Local Search for Bit-Vector Logics in SMT with Path Propagation **53**
1 Introduction . . . 54
2 SLS for QF_BV at a glance . . . 54
3 Propagation Moves . . . 58
4 Experiments . . . 66
5 Conclusion . . . 73

B Propagation Based Local Search for Bit-Precise Reasoning **77**
1 Introduction . . . 78
2 Overview . . . 79
3 Bit-Level . . . 81
4 Word-Level . . . 86
5 Experimental Evaluation . . . 99
6 Conclusion . . . 111

C Turbo-Charging Lemmas on Demand with Don't Care Reasoning **115**
1 Introduction . . . 116
2 Lemmas on Demand at a Glance . . . 118
3 Partial Model Extraction . . . 118
4 Experimental Evaluation . . . 126
5 Conclusion . . . 133

D ddSMT: A Delta Debugger for the SMT-LIB v2 Format **137**
1 Introduction . . . 138
2 The Delta Debugger ddSMT . . . 139
3 Experimental Evaluation . . . 149
4 Conclusion . . . 150

Bibliography **151**

Part I

Prologue

Chapter 1

Introduction

In the field of hardware and software verification, many applications require more expressive logics than propositional logic. Frequently, the most natural way to formulate their problems is to translate them into first-order logic with respect to some background theories, also referred to as Satisfiability Modulo Theories (SMT), which allows reasoning within certain domains. As a consequence, they typically rely heavily on SMT solvers, which provide efficient and highly specialized procedures tailored to the background theories in question. The majority of verification applications requires bit-precise reasoning, often combined with arrays and uninterpreted functions. Fixed-size bit-vectors provide a natural way to reason about circuits and programs, while arrays allow to model memory and actual array data structures. Uninterpreted functions, on the other hand, can be useful as an abstraction for irrelevant or too complex details of a system.

In this thesis, we focus on SMT procedures for bit-vector logics, in particular for the quantifier-free theory of fixed-size bit-vectors. Current state-of-the-art procedures for determining the satisfiability of quantifier-free bit-vector formulas in SMT typically rely on a flattening technique generally referred to as *bit-blasting* (e.g., [80]), where the input formula is eagerly translated into propositional logic and delegated to a SAT solver. However, as shown in [79], translating a quantifier-free bit-vector formula into SAT is in general exponential in the formula size, and deciding the SMT problem for the quantifier-free theory of fixed-size bit-vectors is in general NEXPTIME-complete. While bit-blasting is efficient in practice, its performance entirely depends on sufficiently simplifying the input prior to handing it to the underlying SAT solver, and in particular for increasing bit-widths it usually does not scale well. In this thesis, we explore and propose alternative approaches for solving quantifier-free bit-vector formulas based on local search that do not require bit-blasting and yield a substantial gain in performance in a sequential portfolio combination with bit-blasting.

Bit-blasting is a typical example of *eager* SMT techniques, which encode an SMT formula into an equisatisfiable propositional formula and hand it to a SAT solver back-end. So-called *lazy* SMT approaches, on the other hand, usually tightly integrate a SAT solver to enumerate truth assignments of an abstraction of the input formula. These assignments are then checked for consistency by one ore more theory solvers, which guide the SAT solver through its

search. The majority of current state-of-the-art SMT solvers employ lazy procedures [11,42,49,53,91]. In this thesis, we propose an optimization of such a lazy technique, the Lemmas on Demand (LOD) procedure as introduced in [96] for the quantifier theory of fixed-size bit-vectors with arrays and uninterpreted functions, extended by non-recursive first-order lambda terms. Lemmas on Demand as in [96] is an abstraction refinement technique similar to Counterexample-Guided Abstraction Refinement (CEGAR) in the context of model-checking [43] and refines full candidate models of a bit-vector abstraction of the input formula with lemmas until convergence. Checking the consistency of a full candidate model, however, may introduce unnecessary overhead since it may include parts of the input formula that are irrelevant under the current assignment. By focusing refinement on relevant parts of the input formula, our optimization considerably improves performance of the LOD procedure.

We implemented all our techniques in our SMT solver Boolector [91], which contributed to winning several tracks of the SMT competitions 2014, 2015 and 2016. Boolector is a specialized SMT solver for the quantifier-free theories of fixed-size bit-vectors, arrays and uninterpreted functions. It further natively handles non-recursive first-order lambda terms. Unsurprisingly, Boolector is a complex piece of software and consists of more than 80k lines of code, our testing framework and testing tools not included. Since SMT solvers usually serve as back-end to some application, the level of trust in this application strongly depends on the level of trust in the underlying SMT solver. As a consequence, the key requirements for SMT solvers are correctness, robustness and high performance. To ensure correctness and robustness of an SMT solver, solver developers usually rely on traditional testing techniques such as unit tests and regression testing. However, due to the complex nature of SMT solver engines, this alone is often insufficient [32]. In this thesis, we address automated testing and debugging techniques integrated into the development process of Boolector that we consider crucial for SMT solver development in general.

1.1 Outline and Contributions

This thesis consists of two parts. The second part includes four peer-reviewed Papers A-D of which the author of this thesis is the main author. These papers have been included as originally published, with the exception of using a consistent layout and bibliography, and introducing minor fixes (as explicitly stated at the beginning of each paper chapter) that do not affect the content.

The first part of this thesis is structured as follows. In Chapter 2 we first give a high-level introduction to the background of this thesis. In Chapters 3-7 we then introduce, discuss and extend the contributions of Papers A-D.

Paper A [93] *Improving Local Search for Bit-Vector Logics in SMT with Path Propagation* with Mathias Preiner, Armin Biere and Andreas Fröhlich. In Proceedings of the 4th International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS 2015), affiliated to the 15th International Conference on Formal Methods in Computer Aided Design (FMCAD 2015), 10 pages, Austin, TX, USA, 2015.

In Paper A we first reimplement the SLS for SMT approach of [58] in our SMT solver Boolector, and then improve this approach by introducing an additional propagation-based strategy that takes full advantage of the word-level structure. The basic idea to extend the technique in [58] with a propagation-based strategy was a result of discussions of all authors. Based on this idea, A. Niemetz developed the propagation-based strategy proposed in Paper A. All techniques in Paper A were described and implemented in Boolector by A. Niemetz. The experimental analysis was performed and described by A. Niemetz. The co-authors further contributed with discussions and proof reading Paper A.

Paper B (extends [92]) *Propagation Based Local Search for Bit-Precise Reasoning* with Mathias Preiner and Armin Biere. Accepted for the Special Issue on Recent Topics in Satisfiability Modulo Theories of the International Journal on Formal Methods in System Design (FMSD), to appear. This paper is an extended version of [92], published in Part I of the Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016), pages 179–186, Toronto, ON, Canada, 2016. Paper B as included in this thesis is a preprint version and may slightly differ from the final version.

In Paper B, we present a simple and complete propagation-based local search variant of the procedure proposed in Paper A. The procedure in Paper B was developed and implemented in Boolector by A. Niemetz. The completeness proof presented in Paper B was a joint effort of A. Niemetz and A. Biere. A. Biere further contributed an example to illustrate that focusing on inverse values only when down-propagating assignments is incomplete, and as a result, A. Niemetz and A. Biere developed the notion of consistent values. The techniques in Paper B were mainly described by A. Niemetz, with contributions of A. Biere. The experimental evaluation was performed and described by A. Niemetz, with contributions by M. Preiner. The co-authors further contributed with discussions and proof reading Paper B.

Paper C [90] *Turbo-Charging Lemmas on Demand with Don't Care Reasoning* with Mathias Preiner and Armin Biere. In Proceedings of the 14th International Conference on Formal Methods in Computer Aided Design (FMCAD 2014), pages 179–186, Lausanne, Switzerland, 2014.

In Paper C we introduce an optimization of the lemmas on demand (LOD) procedure as implemented in Boolector, which improves performance by focusing on relevant parts of an inconsistent candidate model for refinement,

only. We propose a justification-based and a dual-propagation-based strategy and compare the performance of both techniques. The basic idea of our dual-propagation-based strategy was developed by A. Niemetz and M. Preiner. The dual-propagation-based procedure was developed based on this idea and implemented in Boolector by A. Niemetz. A. Biere contributed the idea to compare our dual-propagation-based strategy to a justification-based approach. Our justification-based strategy was then developed by A. Niemetz and M. Preiner, and implemented in Boolector by M. Preiner. All techniques in Paper C were described by A. Niemetz. The experimental evaluation was run by M. Preiner and the experimental results were analysed and described by A. Niemetz. The co-authors further contributed with discussions and proof reading Paper C.

Paper D [89] *ddSMT: A Delta Debugger for the SMT-LIB v2 Format* with Armin Biere. In Proceedings of the 11th International Workshop on Satisfiability Modulo Theories (SMT'13), affiliated to the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT 2013), pages 36–45, Helsinki, Finland, 2013.

In Paper D we present ddSMT, a delta debugger for the SMT-LIB v2.0 language, which aims to overcome limitations of previous delta debugging techniques for the SMT-LIB v1 language [32]. The tool and all techniques in Paper D were developed, implemented and described by A. Niemetz. The experimental evaluation was performed and described by A. Niemetz. The co-author contributed with discussions and proof reading Paper D.

Chapter 2

Background

Since Cook proved in 1971 that the satisfiability problem of propositional logic (SAT) is NP-complete [46], it has been established as *the* classical NP-complete problem and is considered as one of the most fundamental problems of computer science. Assuming that $P \neq NP$, procedures to solve the SAT problem (so-called SAT solvers) may therefore have exponential runtime in the worst case. However, in practice, state-of-the-art SAT solvers are efficient and easily tackle industrial-strength problems involving millions of variables.

For many applications, in particular in the field of formal methods for hardware and software development, it is required to formulate the input problem in more expressive logics than SAT. These applications are typically interested in satisfiability of first-order logic (FOL) with respect to some background theories, also referred to as Satisfiability Modulo Theories (SMT), where the background theories fix the interpretation of certain predicate and function symbols. The problem of satisfiability of FOL is in general undecidable, however, some (fragments of) first-order theories are efficiently decidable. In this thesis, we focus on decision procedures for one such decidable fragment, the quantifier-free theory of fixed-size bit-vectors (often combined with arrays and uninterpreted functions), which is of particular importance since it provides a natural way to reason about circuits, memory, programs and array data structures. Procedures for solving SMT, so-called SMT solvers, are efficient in practice and serve as back-end for applications in academia and industry. The most prominent example is the whitebox fuzz testing tool SAGE at Microsoft [62]. Other use cases include symbolic execution and test case generation (e.g., [40, 61, 78, 101]), model checking (e.g., [9, 69]), extended static checking and program verification (e.g., [1, 10, 44, 45, 56]), compiler verification (e.g., [84, 85]) and theorem proving (e.g., [27]). This chapter gives a high level introduction to SAT and SMT and provides an overview of the background of this thesis.

2.1 SAT

Propositional logic, or Boolean logic, reasons about the truth values of propositions and is defined over Boolean variables, the Boolean operators negation ($\neg$) and conjunction ($\wedge$), and the Boolean constants *true* (1 or $\top$) and *false* (0 or $\bot$).

All other Boolean operators can be constructed using $\{\neg, \wedge\}$, e.g., a disjunction over two variables a and b can be expressed as $a \vee b = \neg(\neg a \wedge \neg b)$.

Given a Boolean formula ϕ defined over the Boolean variables $\{a_1, \ldots, a_n\}$, the SAT problem is to decide if there exists an assignment of truth values to the variables such that $\phi(a_1, \ldots, a_n) = 1$. If this is the case, we say that ϕ is *satisfiable* and refer to the set of truth assignments that satisfy ϕ as *satisfying assignment* or *model*. Else, we say that ϕ is *unsatisfiable*.

Example 2.1. As an example, consider formula $\phi_1 = (\neg a \Leftrightarrow b) \wedge (a \Rightarrow b)$. A satisfying assignment for ϕ_1 is $\{a \mapsto 0, b \mapsto 1\}$. However, if we rule out assignment $b \mapsto 1$, we yield the unsatisfiable formula $(\neg a \Leftrightarrow b) \wedge (a \Rightarrow b) \wedge \neg b$.

A majority of decision procedures for SAT expect the input formula to be in conjunctive normal form (CNF), which is a conjunction of disjunctions defined over the Boolean operators $\{\neg, \wedge, \vee\}$. Formula ϕ_1, e.g., can be directly translated into CNF by expressing $(\neg a \Leftrightarrow b)$ as $(a \Rightarrow b) \wedge (b \Rightarrow a)$, which then yields $\phi_1 = (a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg a \vee b)$. Note that any arbitrary Boolean formula defined over the Boolean operators $\{\neg, \wedge, \vee\}$ can be converted into CNF with at most linear growth in formula size via Tseitin transformation by introducing auxiliary variables for the Boolean operators [102].

The majority of current state-of-the-art SAT procedures are based on Conflict-Driven Clause Learning (CDCL) [100], which is a powerful extension of the resolution-based Davis-Putnam-Logemann-Loveland (DPLL) procedure [47]. In this thesis, we will not directly address procedures for SAT. However, SAT solvers, and in particular CDCL-based SAT solvers, often serve as back-end to SMT procedures. For a more in depth introduction to SAT and SAT solving techniques, refer to the *Handbook of Satisfiability* [25] or Knuth's Chapter 7.2.2.2. of *The Art of Computer Programming* on Satisfiability [77].

2.2 SMT

First-order logic extends propositional logic with quantifiers, first-order variables, and predicate and function symbols. We focus on quantifier-free fragments of FOL, in particular the first-order theories of quantifier-free fixed-size bit-vectors in the context of SMT. In the following, we adopt the notions of FOL and first-order theories as defined in [25, 28] and the SMT-LIB standard v2 [13].

A first-order theory $\mathcal{T}$ is defined by a signature Σ and a set of axioms $\mathcal{A}$. A signature Σ is a set of predicate and function symbols, and since FOL extends propositional logic, Σ also includes propositional symbols. All symbols in Σ are associated with a sort and an arity greater or equal 0, e.g., the Boolean operator $\neg$ has sort Bool and arity 1. We generally refer to 0-arity symbols as constants, and in the context of quantifier-free theories we call 0-arity propositional symbols *Boolean variables*. The axioms in $\mathcal{A}$ are constructed from symbols in Σ and define the *interpretation* of all non-constant symbols in Σ. We refer to symbols that

are not in Σ as *uninterpreted* symbols. Note that Σ is not necessarily finite, e.g., the theory of fixed-size bit-vectors has an infinite signature.

Given a quantifier-free Σ-formula ψ, i.e., a formula constructed from symbols in Σ, the SMT problem is to decide if there exists an assignment of domain values to its constant, predicate and function symbols such that ψ evaluates to *true* under the interpretation of theory $\mathcal{T}$. If this is the case, we say that formula ψ is $\mathcal{T}$*-satisfiable* (or *satisfiable*), and else $\mathcal{T}$*-unsatisfiable* (or *unsatisfiable*). We refer to the set of assignments that satisfy ψ as a $\mathcal{T}$*-model* (or *model*) of ψ.

Procedures for SMT are referred to as SMT solvers and are usually divided into so-called *eager* and *lazy* approaches [99]. Eager approaches usually apply various theory-level simplification techniques before translating a given Σ-formula into an equisatisfiable propositional formula, which is then handed to a SAT solver. As a consequence, they can only be applied to theories that can be reduced to propositional logic, at the cost of a possibly considerable blow-up in formula size. The most prominent example of an eager approach is *bit-blasting*, e.g., [80], which is currently state-of-the-art for the theory of quantifier-free fixed-size bit-vectors. While efficient in practice, bit-blasting may suffer from an exponential blow-up [79] and usually does not scale well for increasing bit-widths. Note that in the past it has often been assumed that the SMT problem for the quantifier-free theory of fixed-size bit-vectors as defined in the SMT-LIB standard is NP-complete. However, it is actually NEXPTIME complete [79].

In Paper A and B we will discuss alternative approaches based on *local search* that solve bit-vector problems on the theory-level and do not require bit-blasting or an underlying SAT solver. Starting with a random but complete initial assignment, local search procedures aim to locally improve the current state until a solution is found. As a consequence, they are in general incomplete in the sense that it is not possible to determine unsatisfiability. In Paper A we propose a propagation-based extension of an existing stochastic local search (SLS) procedure for SMT [58], which lifts SLS for SAT (e.g., [72]) from the bit-level to the theory-level and mainly relies on "guessing" guided by heuristics. A more detailed introduction to SLS for SMT is given in Paper A.

In contrast to eager SMT approaches, which directly translate an input problem into SAT, lazy techniques usually iteratively refine an abstraction of the input formula and tightly integrate a SAT solver and one or more theory solvers. The SAT solver typically enumerates truth assignments of a Boolean formula abstraction while the theory solvers check these assignments for consistency and guide the SAT solver through its search. The majority of state-of-the-art SMT solvers implement lazy approaches [11, 42, 49, 53, 91]. They are usually either based on the DPLL(T) framework [94], which lifts the DPLL procedure from SAT to SMT, or employ an abstraction refinement technique similar to the Counterexample-Guided Abstraction Refinement (CEGAR) approach for model checking introduced in [43].

Operator	SMT-LIB	Arity	Bit-Width Output	Input			
$-$	`bvneg`	1	w	w	-	-	Two's complement
$\sim$	`bvnot`	1	w	w	-	-	Bit-wise negation
$[j:i]$	`extract`	1	$j-i+1$	w	w	-	Extraction ($w \geq j \geq i \geq 1$)
$=$	`=`	2	1	w	w	-	Equality
&	`bvand`	2	w	w	w	-	Bit-wise conjunction
$<$	`bvult`	2	1	w	w	-	Unsigned less than
$\ll$	`bvlshl`	2	w	w	w	-	Logical shift left
$\gg$	`bvlshr`	2	w	w	w	-	Logical shift right
$+$	`bvadd`	2	w	w	w	-	Addition
$\cdot$	`bvmul`	2	w	w	w	-	Multiplication
$\div$	`bvudiv`	2	w	w	w	-	Unsigned division
mod	`bvurem`	2	w	w	w	-	Unsigned remainder
$\circ$	`concat`	2	$p+q$	p	q	-	Concatenation
if-then-else	`ite`	3	w	1	w	w	Conditional

Table 2.1: The set of bit-vector operators in Σ_{BV}.

In Paper C, we will focus on an optimization of such a lazy technique, the Lemmas on Demand (LOD) procedure for the quantifier-free theories of fixed-size bit-vectors, arrays and uninterpreted functions [96]. Lemmas on Demand as in [96] is a CEGAR-style SMT procedure that enumerates truth assignments of a bit-vector abstraction of the input formula and refines this assignments with lemmas until convergence. A more detailed overview of the LOD procedure in [96] is given in Paper C.

2.2.1 The Theory of Fixed-Size Bit-Vectors

A fixed-size bit-vector is defined as a finite sequence of bits of length $w \in \mathbb{N}$. The length of a bit-vector is usually referred to as *bit-width* and bit-vectors with different widths correspond to different sorts. We call 0-arity bit-vector symbols *bit-vector variables* and bit-vector values *bit-vector constants*, and usually represent bit-vector constants as binary numbers. Further, for the sake of simplicity we interpret Boolean expressions as bit-vector expressions of bit-width one and define bit indices as starting with 1 rather than 0.

A bit-vector expression n of width w is denoted as $n_{[w]}$ and we will omit the subscript if the context allows. We refer to the i-th bit of $n_{[w]}$ as $n_{[w]}[i]$ (or simply $n[i]$) and denote bit ranges from index i to j as $n[j:i]$ with $i \leq j$. We interpret $n[1]$ as the least significant bit (LSB) and $n[w]$ as the most significant bit (MSB) of $n_{[w]}$, and when representing a bit-vector value as binary

number we identify the far left bit as the MSB and the far right bit as the LSB. Note that we sometimes use decimal values for bit-vector constants as short hand, e.g., $2_{[4]}$ corresponds to the binary value 0010.

We define the quantifier-free theory of fixed-size bit-vectors $\mathcal{T}_{\mathcal{BV}}$ according to the SMT-LIB standard v2 [13, 14]. The signature Σ_{BV} of theory $\mathcal{T}_{\mathcal{BV}}$ is infinite and introduces the set of interpreted bit-vector predicate and function symbols listed in Table 2.1. For convenience, bit-vector logics usually introduce additional bit-vector operators, e.g., signed alternatives to the predicate and function symbols in Table 2.1. All bit-vector operators not listed in Table 2.1 can be expressed with the symbols defined in Σ_{BV}.

Note that the semantics of arithmetic bit-vector operators of bit-width w correspond to the semantics of the respective arithmetic operators in $\mathbb{N}$ modulo 2^w, e.g., $1_{[2]} + 3_{[2]} = 2_{[2]}$. Further, rather than introducing uninterpreted functions, we define an unsigned division $\div_{[w]}$ by 0 to return the greatest possible value $2^w - 1$, i.e., $n \div 0 = \sim 0$. Similarly, $n \bmod 0 = n$.

2.3 Bit-Blasting in Boolector

Bit-blasting is a flattening technique that eagerly translates a quantifier-free bit-vector formula into an equisatisfiable propositional formula as described in, e.g., [80]. Bit-blasting as implemented in our SMT solver Boolector [91] first constructs an intermediate And-Inverter-Graph (AIG) [81] circuit representation of a quantifier-free bit-vector formula by mapping each bit-vector operation to a corresponding AIG circuit. Each AIG is simplified via local two-level AIG rewriting as described in [31] during construction. The resulting AIG representation of the bit-vector formula is then converted into CNF via Tseitin transformation and handed to the underlying SAT solver. Boolector currently supports Lingeling [23], PicoSAT [20] and MiniSat [55] as SAT solver back-end.

In this thesis, we will not directly address bit-blasting techniques. However, bit-blasting as implemented in Boolector will serve as a reference for the local search approaches presented in Papers A and B.

Chapter 3

Paper A: Improving Local Search for Bit-Vector Logics in SMT with Path Propagation

The most common approach for deciding SMT for the quantifier-free theory of fixed-size bit-vectors is bit-blasting, usually in combination with sophisticated rewriting and simplification techniques to simplify the input before eagerly reducing it to propositional logic. While bit-blasting is current state-of-the-art and efficient in practice, it may suffer from an exponential blow-up [79] and in general does not scale well for large bit-widths. To avoid this problem, in [58], Fröhlich et al. lifted SLS from the SAT to the SMT level and proposed an SLS procedure for bit-vector logics that does not reduce the input problem to SAT but operates directly on the theory level, with promising initial results. However, their approach mostly simulates bit-level local search by focusing on single bit flips rather than fully exploiting the advantages of working on the theory level. In Paper A we first reimplement the SLS for bit-vector logics approach in [58] in our SMT solver Boolector and confirm its effectiveness. We then improve the technique presented in [58] by introducing an additional propagation-based strategy that takes full advantage of the word-level structure. Our results suggest a combination of our techniques with a state-of-the-art bit-blasting engine within a sequential portfolio setting [104].

3.1 Discussion

In Paper A we observed that initializing the random number generator (RNG) of Boolector with different seeds has almost no influence on the number of solved instances of our SLS configurations. This is, however, not true and was caused by a bug in Boolector where passing a seed via the command line was ignored and instead the default value 0 was used.

In the following, we evaluate randomization effects of our reimplementation of [58] and the propagation-based techniques presented in Paper A. We use the same identifiers for configurations that correspond to a configuration in Paper A and consider the following configurations.

(1) **Bsls** The SLS engine of the current version 2.4 of Boolector, optionally with random walks enabled (+rw).

(2) **Bprop** The SLS engine of the current version 2.4 of Boolector with our propagation-based strategy enabled. This configuration uses propagation moves only, optionally with conflict recovery via random walk (+frw).

We use the same benchmark set as in Paper A, which consists of all 16436 benchmarks with status *sat* and *unknown* of the QF_BV category of the SMT-LIB [14] except those proved by Boolector's bit-blasting engine to be unsatisfiable within 1200 seconds. The current version 2.4. of Boolector, however, determined that 210 of these benchmarks are actually unsatisfiable. In order to be able to compare the results in Paper A with the results in this section, we still include these unsatisfiable instances and use the full benchmark set for our evaluation.

All experiments were run on the same hardware setup as in Paper A (a cluster of 30 nodes of 2.83 GHz Intel Core 2 Quad machines running Ubuntu 14.04.5 LTS) with the same time and memory limits. In case of a time or memory out, a penalty of the given time limit was added to the total CPU time.

In order to evaluate the influence of using different seeds for the RNG of Boolector, we ran a batch of 11 runs per configuration, one with default seed 0 and ten with different random seeds. Figure 3.1 and 3.2 show the results for configurations Bsls, Bsls+rw, Bprop and Bprop+frw in terms of number of solved instances with a time limit of 10 and 1 seconds as box-and-whiskers plots, with the values of the runs with default seed 0 indicated as a red diamond.

Overall, with a time limit of 10 seconds, configuration Bsls outperforms the other three configurations and increases the number of solved instances compared to the results of Paper A by approximately 11%. This is due to an optimization of the score computation of Boolector's SLS engine, which had a huge impact on performance. However, Bsls outperforms our propagation-based strategy only on three benchmark families (sage, Sage2, and stp_samples), and in particular families sage and Sage2 are overrepresented in our benchmark set since they make up more than 80% of its instances. In Chapter 4 we will analyse why the SLS strategy of configuration Bsls performs better than our propagation-based strategy on these particular benchmark families.

As observed in Paper A, enabling random walks (+rw) does not improve performance for configuration Bsls, and using random walks rather than regular SLS moves when recovering from a conflict (+frw) for configuration Bprop does not increase the number of solved instances within ten seconds but delivers the best results of all four configurations with a time limit of one second.

In terms of randomization effects, with an inter-quartile range (IQR) of 75 instances and a median absolute deviation (MAD) of 53.4, configuration Bsls seems to be the least robust with respect to randomization effects within a time limit of ten seconds. Our propagation-based strategy Bprop, on the other hand, seems to be least affected with an IQR of 24 instances and a MAD of 22.2. When de-

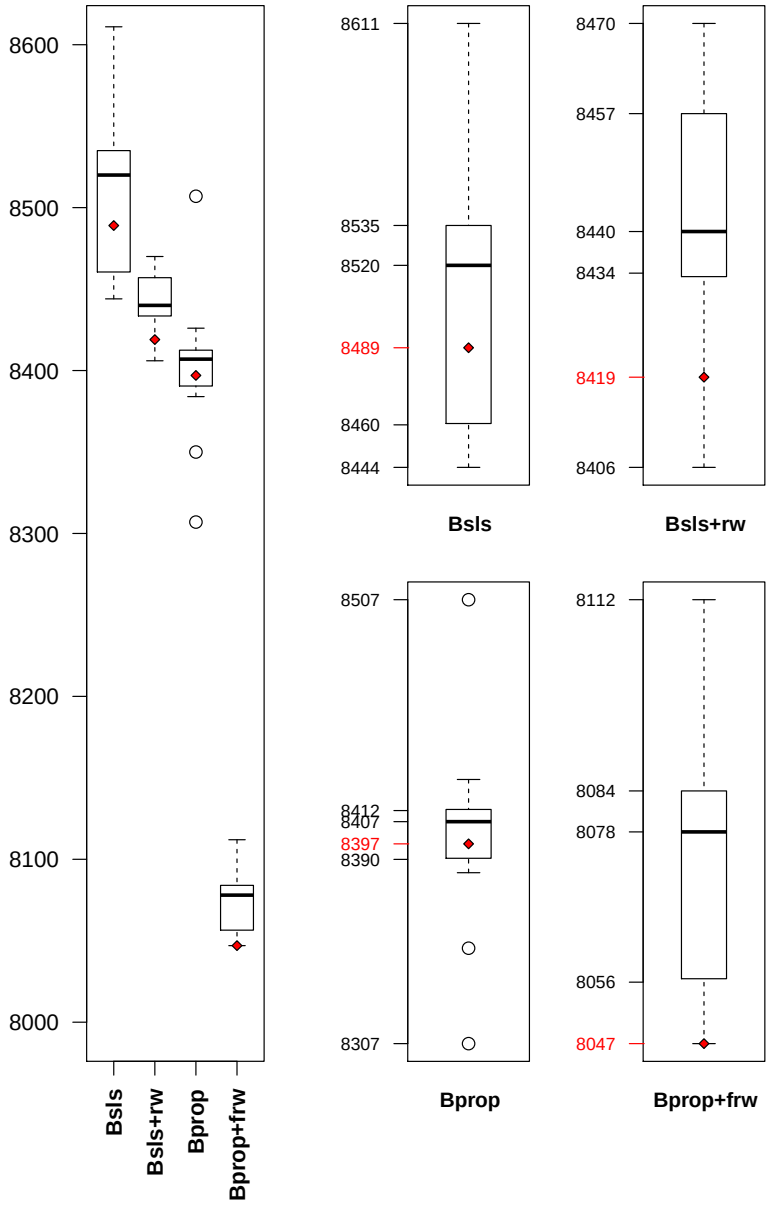


Figure 3.1: Number of solved instances and randomization effects over 11 runs of configurations Bsls, Bsls+rw, Bprop and Bprop+frw with different seeds for the RNG and a time limit of 10 seconds.

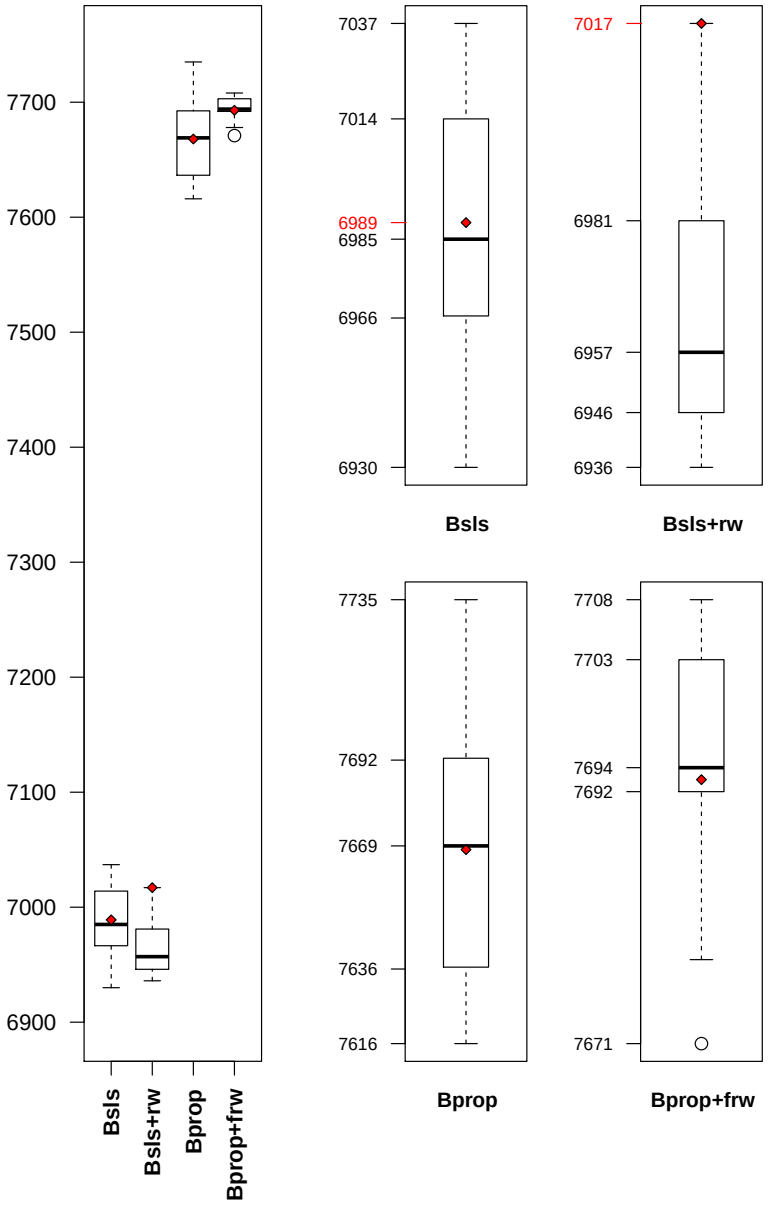


Figure 3.2: Number of solved instances and randomization effects over 11 runs of configurations Bsls, Bsls+rw, Bprop and Bprop+frw with different seeds for the RNG and a time limit of 1 second.

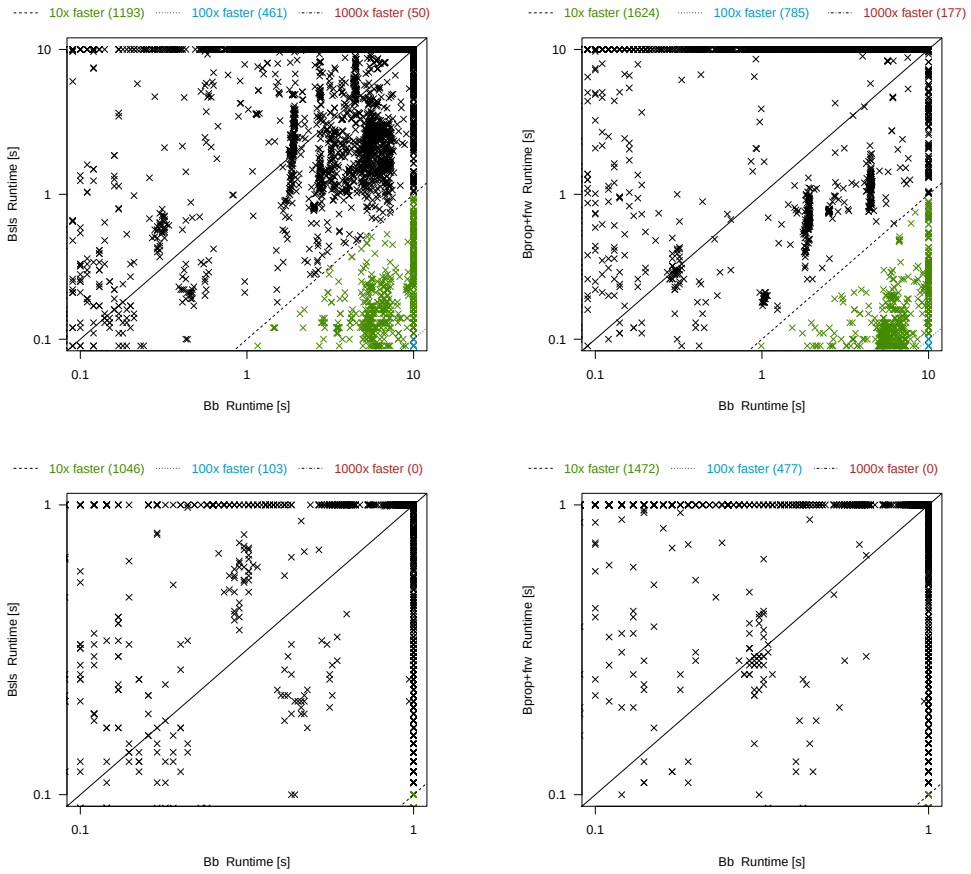


Figure 3.3: Runtime comparison of Bsls and Bprop+frw with the bit-blasting configuration Bb with a time limit of 10 (top) and 1 (bottom) seconds. Note that for configurations Bsls and Bprop+frw we used the runs with default seed 0.

creasing the time limit to one second, the configuration most affected is Bprop (with an IQR of 56 instances and a MAD of 51.9) and the configuration least affected is Bprop+frw (with an IQR of 11 instances and a MAD of 11.9).

In terms of runtime, as illustrated in Figure 3.3, both when considering a time limit of 10 and of 1 seconds, compared to the bit-blasting engine Bb our propagation-based strategy Bprop+frw shows a considerably better improvement in performance than configuration Bsls. This is in particular interesting since Bsls solves considerably more instances than Bprop+frw within a time limit of 10 seconds (albeit with a time limit of 1 seconds it is vice versa).

As concluded in Paper A, these results suggest that Bprop+frw is the best choice for a sequential portfolio combination with bit-blasting where we assume that Bprop is run for one second prior to falling back to the bit-blasting engine. In Paper A, we performed a virtual experiment simulating such a combination,

with promising results. Since then we implemented a sequential portfolio configuration in Boolector that allows to combine the SLS-based techniques described in Paper A with the bit-blasting engine of Boolector. However, since using actual runtime as a limit for the strategy run prior to bit-blasting is unreliable for practical reasons, we use metrics that do not require to measure time. For our evaluation, we consider the following configurations.

(1) **Bb** The bit-blasting engine of the current version 2.4 of Boolector.

(2) **Bb+Bsls-virtual-Xs** A virtual sequential portfolio combination of Bb and Bsls where we assume that configuration Bsls is run exactly X seconds prior to invoking Bb.

(3) **Bb+Bsls-X** The sequential portfolio combination of Bsls and Bb as implemented in Boolector 2.4 where configuration Bsls is run with a limit of X explorations of neighboring states.

(4) **Bb+Bprop+frw-virtual-Xs** A virtual sequential portfolio combination of Bb and Bprop+frw where we assume that Bprop+frw is run exactly X seconds prior to invoking Bb.

(5) **Bb+Bprop+frw-X** The sequential portfolio combination of Bprop+frw and Bb as implemented in Boolector 2.4 where configuration Bprop+frw is run with a limit of X propagations.

We run all our experiments with our sequential portfolio combinations with a time limit of 1200 seconds. Further, we used configurations Bsls and Bprop+frw with default seed 0 for the RNG.

Figure 3.4 compares the performance of the bit-blasting configuration Bb with the virtual configuration Bb+Bsls-virtual-1s and the sequential portfolio combinations Bb+Bsls-1k, Bb+Bsls-10k and Bb+Bsls-100k with a time limit of 1200 seconds. Overall, a virtual configuration Bb+Bsls-virtual-1s improves runtime by 4% and solves 31 instances more than Bb. Within a real sequential portfolio setting, configuration Bb+Bsls-10k shows the best performance and even outperforms Bb+Bsls-virtual-1s in the number of solved instances (+53 compared to Bb). However, in terms of runtime it introduces too much overhead for instances that can not be solved by Bsls within the given limit which implies that using the number of explored neighboring states as a limit is not good enough. Finding a better metric as a limit for Bsls is left to future work.

Figure 3.5 compares the runtime of Bb and the virtual and actual sequential portfolio combinations Bb+Bprop+frw-virtual-1s and Bb+Bprop+frw-10k with a time limit of 1200 seconds. We chose 10k propagation steps as a limit for configuration Bb+Bprop+frw-X since in Paper B this delivered the best results for a sequential portfolio combination of Bb and the propagation-based local search procedure presented in Paper B. Overall, a virtual combination Bb+Bprop+frw-

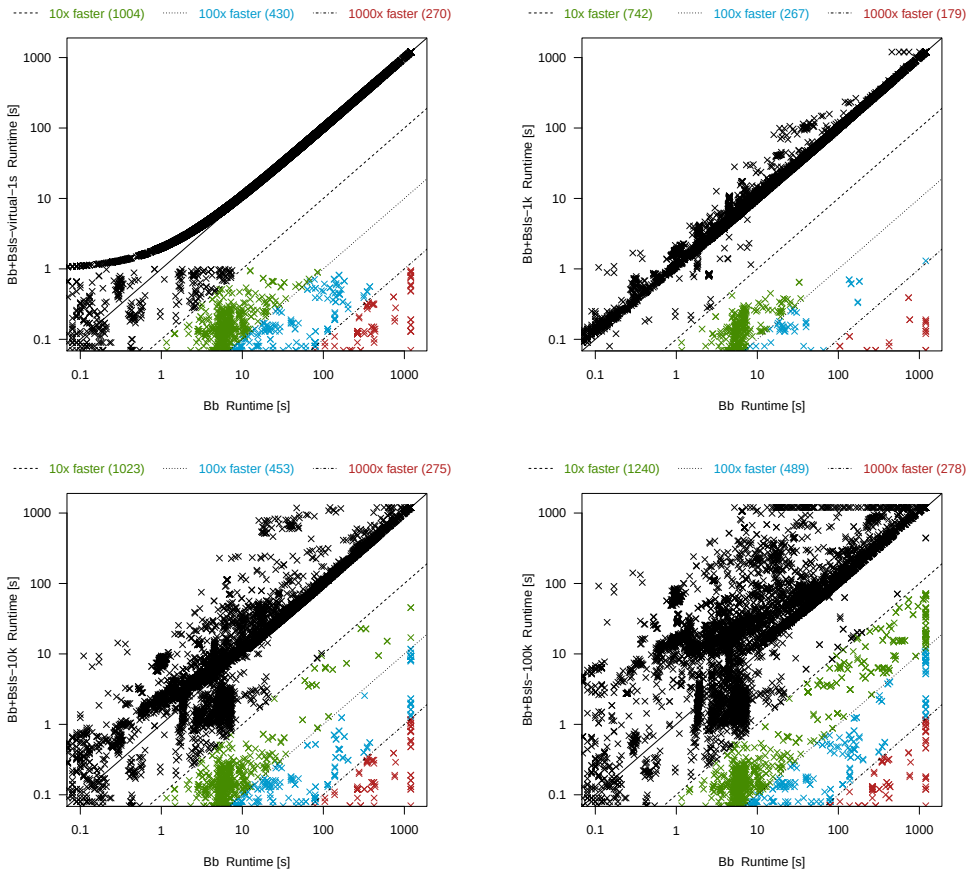


Figure 3.4: Runtime comparison of Bb versus Bb+Bsls-virtual-1s, Bb+Bsls-1k, Bb+Bsls-10k and Bb+Bsls-100k with a time limit of 1200 seconds.

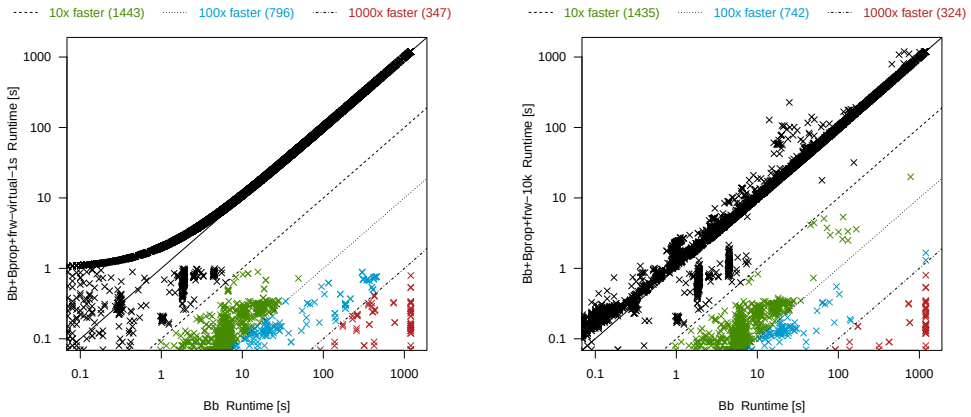


Figure 3.5: Runtime comparison of Bb versus configurations Bb+Pw-virtual-1s and Bprop+frw with a time limit of 1200 seconds.

virtual-1s improves runtime by 6% and solves 51 instances more than Bb. A real sequential portfolio combination Bb+Bprop+frw-10k comes very close to the virtual simulation, with a runtime improvement of more than 5% and 52 more solved instances than Bb. In comparison to Bb+Bsls-10k, Bb+Bprop+frw-10k solves one instance less but improves runtime performance by more than 3%.

Note that our propagation-based approach as presented in Paper A still relies on brute-force randomization and restarts as in [58] to achieve completeness. Further, it can get stuck when down-propagation assignments, in which case it has to fall back on regular SLS moves (or random walks as in configuration Bprop+frw) to recover. This is due to the fact that inverse computation as in Paper A is too restrictive since it introduces short cuts for some operators, e.g., when choosing the simplest rather than some random but valid solution. Relying on inverse computation alone is actually too restrictive in general and may inadvertently prune the search space. In Paper B, we propose a complete propagation-based local search technique that simplifies and extends our approach in Paper A and avoids this problem. Further, it neither relies on regular SLS moves nor brute-force randomization or restarts to achieve completeness.

Chapter 4

Paper B: Propagation Based Local Search for Bit-Precise Reasoning

In Paper A, we extended SLS for bit-vector logics in SMT as presented in [58] with an additional strategy based on propagating assignments from the outputs to the inputs. This significantly improves performance. However, to achieve completeness we still have to rely on brute-force randomization and restarts as in [58]. Further, down-propagating assignments as in Paper A can get stuck, in which case we have to fall back on the SLS techniques described in [58], and focusing on inverse value computation only when down-propagating assignments is too restrictive in general since it may inadvertently prune the search.

In Paper B, we present a simple and complete propagation-based local search variant of the procedure proposed in Paper A. Our approach relies entirely on propagating assignments and does not need to fall back on SLS techniques, brute-force randomization or restarts to achieve completeness. To decide on propagation paths, we extend the notion of bit-level observability don't cares as in the context of Automatic Test Pattern Generation (ATPG) [82] by introducing the notion of *essential* inputs, which lifts the concept of controlling inputs to the word-level. To down-propagate assignments, we formalize the ATPG concept of *backtracing* and lift it from the bit-level to the word-level, which overcomes the problem of too restrictive inverse value computation. We implemented our propagation-based local search strategy and combine it in a sequential portfolio combination with bit-blasting in our SMT solver Boolector. Our experimental results show that our techniques yield a substantial gain in performance. Note that we first presented our propagation-based local search procedure in [92]. Paper B is an extended and revised version of [92].

4.1 Discussion

The results in Paper B show that overall, our propagation-based strategy (configuration Pw) outperforms the SLS for SMT approach as introduced in [58] and implemented in Boolector (configuration Bsls). However, we observed that

in comparison to Bsls, configuration Pw seems to struggle on certain instances of the sage, Sage2 and stp_samples benchmark families. In Paper B we identified 461 such benchmarks, and since recently introducing an optimization of the score computation of Bsls which considerably improved its performance (see Chapter 3) this number even increased to 777. Note that families sage and Sage2 are overrepresented since they make up more than 80% of our benchmark set.

In Paper B we concluded that the better performance of Bsls on these particular benchmarks seems to be due to the fact that Pw is oblivious to bits that can be simplified to constant values since it propagates target values towards the inputs. Bsls, on the other hand, implicitly considers such constant bits since it explores neighboring states of the current assignment by flipping input bits and then computing a score that determines if a certain move brings the current assignment closer to a satisfying assignment. As an example consider an expression $0_{[4]} \circ x_{[4]}$ where the first four resulting bits are constant 0. Configuration Bsls will try to flip bits of bit-vector variable x and will then decide on the best move based on the resulting score. Our propagation-based strategy Pw, on the other hand, may propagate a target value towards this expression that can never be assumed, e.g., 11001111, in which case it will decide on move $x = 1111$ (since 1111 is a consistent value for x) and continue. However, in this particular situation this seems to be disadvantageous.

In the current version 2.4 of Boolector we introduced a heuristic for the scenario above, where we discard the down-propagated target value with a certain probability if it can never be assumed by a concatenation expression and rather flip a random bit of the current assignment of the non-constant expression (x in our example above). We evaluated the effects of this heuristic by running a batch of 11 runs of our propagation-based configuration Pw with different seeds for the RNG (one with default seed 0 and ten with different random seeds) on the same benchmark set as in Paper B (16436 instances) with the same time and memory limits (10 seconds and 7 GB). Overall, by introducing the heuristic described above the performance of configuration Pw in the number of solved instances increased by more than 1%. This suggests that introducing knowledge on constant bits into our propagation-based strategy will indeed increase performance.

In an additional experiment, we evaluated the models of the 777 benchmarks on which our propagation-based strategy Pw seems to have a disadvantage over configuration Bsls, and we identified an interesting pattern. For 80% out of all 777 instances, the assignment of more than 50% of the inputs was 0, and for more than 30% of the non-zero inputs only one bit was set to 1. Since Bsls starts with an initial assignment where all inputs are set to 0, this means that its focus on single bit flips quickly moves the initial assignment towards a satisfying assignment. For almost 50% of these instances, Bsls required even less than 50 moves. Configuration Pw starts with the same initial assignment as Bsls, however, the majority of these 777 benchmarks contain a considerable amount of expressions with constant bits, and as mentioned above this seems to handicap Pw. These results suggests that for this set of benchmarks the strategy

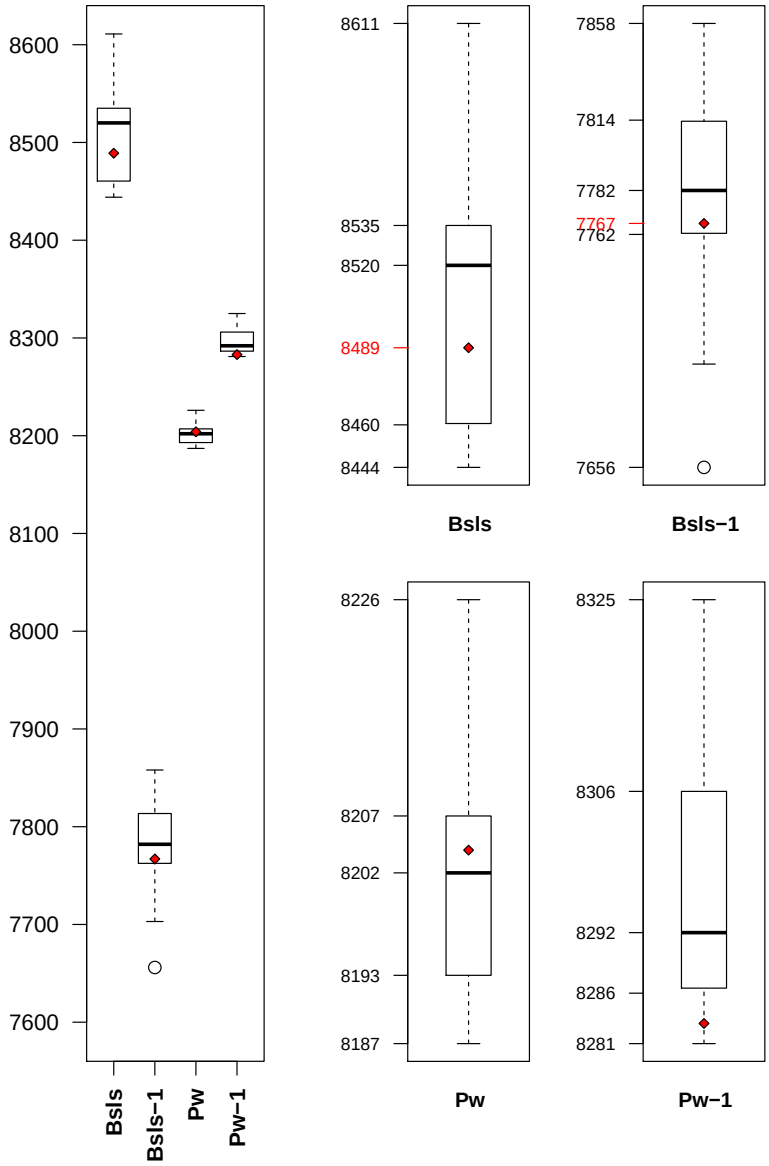


Figure 4.1: Number of solved instances and randomization effects over 11 runs with different seeds for the RNG for each configuration Bsls, Bsls-1, Pw and Pw-1 with a time limit of 10 seconds.

of Bsls is clearly advantageous over Pw and in particular profits from an initial assignment where all inputs are set to 0. Hence, in an additional experiment we initialized the inputs with all bits set to 1 rather than 0 (configurations Bsls-1 and Pw-1) and evaluated the performance of Bsls, Bsls-1, Pw and Pw-1 on the full benchmark set of Paper B over 11 runs with different seeds (again, one with default seed 0 and ten with different random seeds). The results in terms of solved instances within a time limit of 10 seconds are illustrated in Figure 4.1 as box-and-whiskers plots with the values of the runs with default seed 0 indicated as red diamond. Overall, configuration Bsls obviously profits considerably from initializing the inputs with 0 since in comparison to Bsls the performance of Bsls-1 drops by almost 10%. In particular on the set of 777 benchmarks where Bsls had an advantage over our propagation-based strategy Pw, initializing the inputs with 1 resulted in Bsls-1 only solving 97 instances (12.5%) within a time limit of 10 seconds. Our propagation-based strategy, on the other hand, is much more robust than Bsls with respect to the input initialization value and seems to overall even profit from initializing the inputs with 1.

Overall, our results suggest that introducing knowledge on constant bits into our propagation-based approach will indeed considerably increase performance. Further, exploiting knowledge about conflict scenarios as described in the example above may allow to introduce strategies such as lemma generation to obtain an algorithm that allows to also prove unsatisfiability. We leave these improvements to future work.

Chapter 5

Paper C: Turbo-Charging Lemmas on Demand with Don't Care Reasoning

The LOD procedure as implemented in Boolector [93] is a lazy SMT technique for the quantifier-free theories of fixed-size bit-vectors, arrays and uninterpreted functions, which natively and lazily handles non-recursive first-order lambda terms. It is a CEGAR-based model-driven abstraction refinement technique that iteratively enumerates $\mathcal{T}_{\mathcal{BV}}$-models (the candidate models) of a Σ_{BV}-abstraction of the input formula (the bit-vector skeleton) and refines this abstraction until convergence. Each of these candidate models is a complete assignment of the bit-vector skeleton and may include parts of the input formula irrelevant under the current assignment. Consider, e.g., a bit-vector skeleton $x_{[2]} = y_{[2]} \vee a_{[2]} > b_{[2]}$ of some input formula ψ and a candidate model $\{x \mapsto 00,\ y \mapsto 01,\ a \mapsto 01,\ b \mapsto 00\}$, where the consistency of x and y with ψ is irrelevant since $x = y$ is unsatisfied. The cost for abstraction refinement of our LOD procedure, however, directly depends on the number of refinement iterations until convergence, and as a consequence, producing refinements for irrelevant parts of the formula abstraction may introduce unnecessary and costly overhead.

In Paper C we aim to reduce this overhead based on the notion of *observability don't cares* (e.g., [82]), i.e., parts of the formula abstraction irrelevant under the current assignment. Our optimization improves the performance of our LOD procedure by focusing on the relevant parts of an inconsistent candidate model for refinement, only. We employ two different techniques to achieve this goal. One is based on *justification* heuristics as in the context of ATPG [87], and the other is inspired by *dual propagation* techniques in the context of QBF [64, 65]. Both optimizations are competitive and considerably reduce the number of refinement iterations until convergence. They outperform the LOD procedure in [96] and perform equally well in terms of number of solved instances. In terms of runtime, however, our dual-propagation-based technique introduces considerably more overhead than our justification-based technique due to the use of an additional (dual) SMT solver instance. Disregarding the dual solver overhead, our dual-propagation-based technique considerably outperforms our

justification-based technique in terms of runtime. This suggests to adopt a more eager dual propagation approach to render the dual solver overhead negligible, which is rather involved and still left to future work.

5.1 Discussion

The LOD procedure as described in [96] serves as the base procedure for the optimized techniques presented in Paper C. It implements an abstraction refinement loop that generates lemmas lazily with strictly one lemma per iteration, i.e., only for the first inconsistency of a candidate model encountered. A candidate model, however, may contain multiple inconsistent assignments, even though some of them usually directly influence each other. Generating and adding lemmas more eagerly as proposed in [95], i.e., for more than one inconsistency of a candidate model, considerably decreases the number of refinement iterations of the LOD procedure. This consequently improves performance, however, at the possible cost of a considerable increase in the overall number of lemmas generated.

As shown in Paper C, adding lemmas lazily as in [96] may produce a considerable number of iterations where a lemma for a conflict in a part of the formula abstraction irrelevant under the current assignment is generated. Since such don't care conflicts are not considered for abstraction refinement with our justification- and dual-propagation-based techniques, refinement iterations that cover a don't care conflict are essentially skipped entirely. Adding lemmas more eagerly as in [95], on the other hand, increases the probability that a refinement iteration does not only cover don't care conflicts but also conflicts in relevant parts of the formula abstraction. As a consequence, this leaves less room for improvement for the optimization techniques described in Paper C. However, not considering don't care conflicts for abstraction refinement when adding lemmas more eagerly may still be beneficial and increases performance. Further, since the overall number of refinement iterations decreases considerably when adding lemmas more eagerly, this also decreases the overhead introduced by the dual solver instance for our dual-propagation-based optimization.

In the following, we combine the improved lemma generation techniques described in [95] with our justification- and dual-propagation-based optimizations in our SMT solver Boolector and evaluate the following base configurations.

(1) **Btor** The current version 2.4 of Boolector.

(2) **Btor+ju** The current version 2.4 of Boolector with justification-based partial model extraction enabled.

(3) **Btor+dp** The current version 2.4 of Boolector with dual-propagation-based partial model extraction enabled.

As in [95], we distinguish between three different lemma generation strategies, depending on their level of eagerness:

(a) **semi-eager** (default)
The default strategy as implemented in configuration Btor, where in each refinement iteration, lemmas for all independent conflicts of a candidate model up to the first that is influenced by any of the others are added to the formula abstraction.

(b) **eager** (el)
An eager strategy, where lemmas for all conflicts of a candidate model are generated and added to the formula abstraction. We identify configuration Btor with this strategy enabled as Btor_{el} and use Btor_{el}+ju and Btor_{el}+dp for configuration Btor_{el} with our justification- and dual-propagation-based optimization enabled.

(c) **lazy** (ll)
The original lemma generation strategy as described in [96] and employed in Paper C, where in each refinement iteration only the first conflict encountered contributes to abstraction refinement. We identify configuration Btor with this strategy enabled as Btor_{ll} and use Btor_{ll}+ju and Btor_{ll}+dp for configuration Btor_{ll} with our justification- and dual-propagation-based optimization enabled. Note that configuration Btor_{ll} in essence corresponds to an improved version of configuration $\text{Boolector}_{\text{ba}}$ in Paper C.

We evaluate the configurations listed above on the full QF_ABV benchmark set (15091 instances) of the SMT-LIB [14]. Our experiments were run on the same hardware setup (a cluster with 30 nodes of 2.83 GHz Intel Core 2 Quad machines running Ubuntu 14.04.5 LTS) with the same time and memory limit (1200 seconds and 7 GB) as in Paper C. In case of a time or memory out, a penalty of the given time limit was added to the total CPU time.

Table 5.1 summarizes the overall results of base configuration Btor in combination with our justification- and dual-propagation-based optimizations on benchmark set QF_ABV. It lists the number of solved instances (Solved), the number of uniquely solved instances (U), time outs (TO), memory outs (MO), total CPU time (Time), and the dual solver overhead (DS) introduced by our dual-propagation-based optimization, if enabled. Overall, even when generating lemmas more eagerly, enabling our justification-based optimization (Btor+ju) still increases performance, in particular in terms of runtime. This is further illustrated in Figure 5.1a. Our dual-propagation-based optimization Btor+dp, on the other hand, performs slightly worse than Btor and Btor+ju due to the overhead introduced by the dual solver instance, which is mainly responsible for losing instances configuration Btor+ju is able to solve within the given time limit.

Solver	Solved	U	TO	MO	Time [s]	DS [s]
Btor	15047	0	43	1	75076	-
Btor+ju	15049	1	39	3	73801	-
Btor+dp	15044	2	45	2	88067	18314

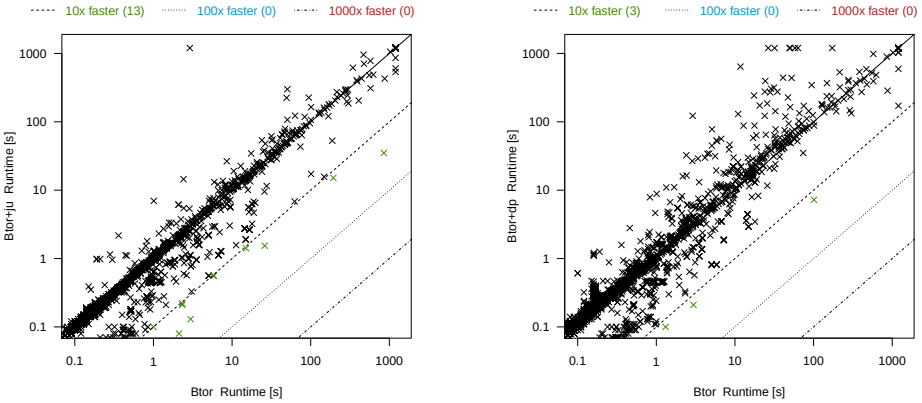
Table 5.1: Overall results of configurations Btor, Btor+ju and Btor+dp on the QF_ABV benchmark set of the SMT-LIB [14].

Solver	Solved	LOD	ITER	Time [s]	DS [s]
Btor	15039	349486	57233	21821	-
Btor+ju	15039	266427	57520	20599	-
Btor+dp	15039	296682	52708	28709	7707

Table 5.2: Results for configurations Btor, Btor+ju and Btor+dp on commonly solved instances of the QF_ABV benchmark set of the SMT-LIB [14].

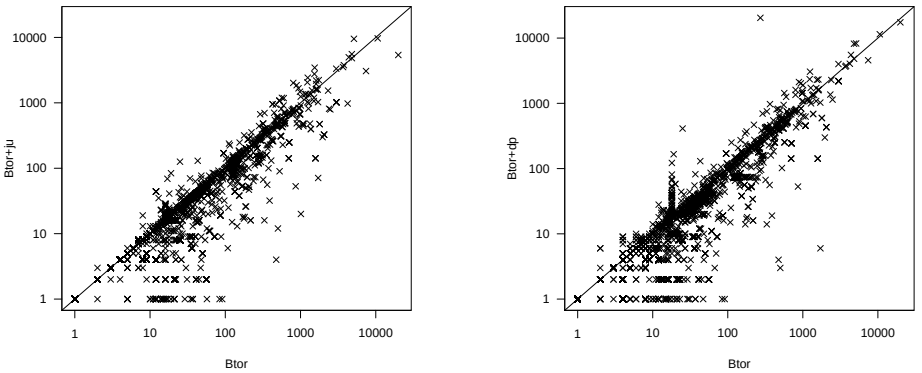
Solver	Solved	LOD	ITER	Time [s]	DS [s]
Btor_{ll}+ju	15037	226428	169995	23985	-
Btor+ju	15037	257943	56775	18890	-
Btor_{el}+ju	15037	315394	45878	21096	-
Btor_{ll}+dp	15007	170979	115598	42170	23824
Btor+dp	15007	197310	46596	19672	3595
Btor_{el}+dp	15007	207418	41036	18729	2646

Table 5.3: Results for our justification- and dual-propagation-based optimizations with different strategies for lemma generation on commonly solved instances of the QF_ABV benchmark set of the SMT-LIB [14].



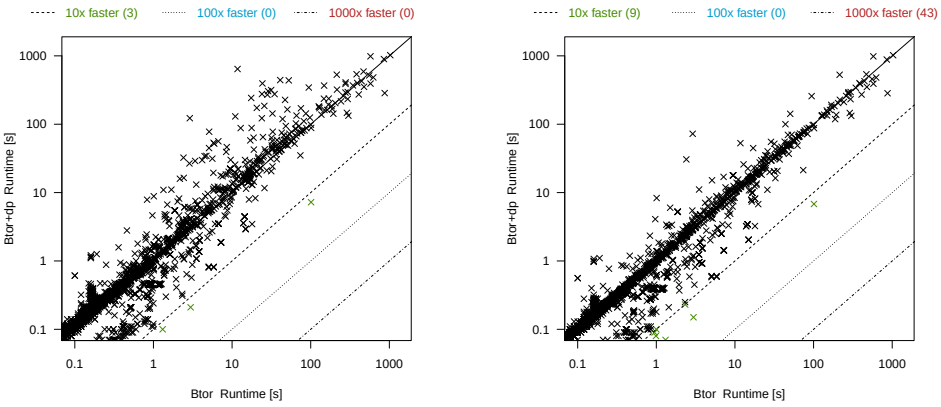
(a) Btor vs. Btor+ju **(b)** Btor vs. Btor+dp

Figure 5.1: Runtime comparison of Btor, Btor+ju and Btor+dp on the full benchmark set QF_ABV with a time limit of 1200 seconds.



(a) Btor vs. Btor+ju **(b)** Btor vs. Btor+dp

Figure 5.2: Comparison of the number of generated lemmas of Btor, Btor+ju and Btor+dp on commonly solved instances of benchmark set QF_BV.



(a) Btor vs. Btor+dp with DS **(b)** Btor vs. Btor+dp without DS

Figure 5.3: Runtime comparison of Btor vs Btor+dp on commonly solved instances of benchmark set QF_BV with and without dual solver overhead (DS).

As illustrated in Figure 5.1b, however, for a considerable number of instances our dual-propagation-based optimization improves runtime despite introducing additional overhead by the dual solver instance.

Table 5.2 illustrates the number of generated lemmas (LOD), the number of refinement iterations (ITER) and the dual solver overhead (DS) for commonly solved instances on set QF_ABV. In comparison to configuration Btor, our justification-based optimization Btor+ju considerably decreases the number of lemmas generated, however, at the cost of an increase in the overall number of refinement iterations. Our dual-propagation-based optimization Btor+dp, on the other hand, not only decreases the number of generated lemmas but also the number of refinement iterations considerably. Figure 5.2 further illustrates the number of generated lemmas of Btor+ju and Btor+dp in comparison to the base configuration Btor on commonly solved instances of the QF_ABV set.

Table 5.3 shows the results for our justification- and dual-propagation-based optimization techniques in combination with different lemma generation strategies for commonly solved instances on the QF_ABV benchmark set. Overall, when generating lemmas (more) eagerly, the number of refinement iterations until convergence drops by 60-75%. In case of our dual-propagation-based optimization, the dual solver overhead even drops by 85-90%. The overhead introduced by the dual solver, however, is still considerable. Figure 5.3 further illustrates this by comparing the runtime of Btor and Btor+dp with and without dual solver overhead on the commonly solved instances of set QF_ABV.

Overall, our results show that generating lemmas more eagerly leaves less room for improvement for our optimization techniques. However, they still improve performance, in particular in terms of runtime. When generating lemmas eagerly, our dual-propagation-based technique introduces a considerable overhead due to the use of an additional dual solver instance. Generating lemmas more lazily significantly decreases this overhead. Introducing techniques to render the dual solver overhead negligible is still left to future work.

Chapter 6

Paper D: ddSMT: A Delta Debugger for the SMT-LIB v2 Format

In Papers A-C, we presented techniques to improve state-of-the-art SMT procedures for the theories of bit-vectors, arrays and uninterpreted functions. Implementing such procedures within an SMT solver is a low-level engineering task, with performance, correctness and robustness as its key requirements. State-of-the-art SMT solvers are typically highly complex pieces of software and since they usually serve as back-end to some application, the level of trust in this application strongly depends on the level of trust in the underlying solver. Full verification of SMT solvers, however, is difficult due to their complex nature and still an open question. To ensure robustness, solver developers therefore usually rely on traditional testing techniques such as unit and regression tests.

In [32], grammar-based black-box fuzz testing has been shown to be effective to uncover bugs in SMT solvers, in particular in combination with delta debugging tools for minimizing failure inducing input. An input fuzzer for some language typically generates random but valid input in this language, and if this input triggers faulty behavior of the system under test, minimizing this input as much as possible while preserving its failure-inducing characteristics by means of delta debugging enables the localization of failure-inducing code in a time efficient manner. The delta debugger DeltaSMT presented in [32], is tailored to quantifier-free logics in the previous version of the SMT-LIB language, the now obsolete SMT-LIB v1 [97]. As a consequence, it is incompatible with the current version SMT-LIB v2 [13], which is a major upgrade of its predecessor. Further limiting is the fact that it employs a hierarchical minimization approach that introduces too much overhead in terms of runtime and may even cause DeltaSMT to not being able to minimize certain input files.

In Paper D, we present ddSMT, a delta debugger for the SMT-LIB v2.0 language [15] that aims to overcome the limitations of DeltaSMT by employing a different algorithmic approach. It provides grammar-based minimization strategies with full support of all SMT-LIB v2 logics, and in particular handles macros, annotations and term-level and command-level scoping. Our results confirm its

effectiveness, and in the development process of Boolector, ddSMT is one of the integral tools of our testing workflow.

Note that the SMT-LIB language is continuously evolving, and as a consequence, ddSMT is a continuous work in progress. Recent changes to the SMT-LIB language as defined in v2.5 of the SMT-LIB standard [13] are currently not yet supported and left to future work.

6.1 Discussion

Delta debugging techniques are automated procedures that enable efficient localization of faulty code by minimizing failure-inducing input. Grammar-based fuzz testing tools have been shown to be effective to uncover bugs in SMT solvers [32], in particular in combination with delta debugging tools such as ddSMT. However, they are entirely input-based.

In [5], model-based API fuzzing for SAT solvers was reported to be more effective than input fuzzing, in particular in combination with option fuzzing. In Chapter 7, we introduce a model-based API testing tool set for our SMT solver Boolector and compare it to the input fuzzing approach presented in [32]. Our results confirm the results in [5].

Chapter 7

BtorMBT: A Model-Based API Tester for Boolector

In [32], grammar-based black-box fuzz testing techniques have been shown to be effective to uncover bugs in SMT solvers, in particular in combination with automated delta debugging procedures for minimizing failure-inducing input. In Paper D, we present such a procedure for the SMT-LIB v2 language [15]. Fuzz testing techniques as in [32] are entirely input-based and, as a consequence, restricted to a certain input language. State-of-the-art SMT solvers, however, usually provide a rich application programming interface (API) as the direct connection between an application and its solver back-end. This API often introduces additional functionality not supported by the input language. As a consequence, by merely generating randomized valid input sequences, it may not be possible to test the full feature set an SMT solver actually provides.

In [5], the authors proposed to apply a model-based testing framework to SAT solvers. This framework randomly generates valid sequences of API calls rather than randomized valid input, and further allows to test all possible valid system configurations by randomly setting and combining configuration options of the solver. In case of an error, an API trace is generated and replayed by a dedicated trace interpreter to reproduce the undesired behavior. Delta debugging techniques reduce an API error trace while preserving its failure-inducing characteristics and enable to locate the cause of the error in a time efficient manner. Applying this approach to the SAT solver Lingeling [23] yields convincing results, and is in particular promising for other solver back-ends.

In this chapter, based on the results in [5] we introduce a model-based API testing framework for our SMT solver Boolector. It consists of the model-based API tester BtorMBT, the trace execution tool BtorUntrace, and the delta debugger ddMBT, and is an integral part of the testing workflow in the development process of Boolector, complemented by basic unit testing, a regression test suite and parser testing tools (e.g., FuzzSMT [30] in combination with ddSMT to test valid input, and the ddsexpr tool set [21] to test for robustness and correct error handling). Since version 1.6, our model-based tester BtorMBT and our trace execution tool BtorUntrace are shipped together with Boolector, and in particular BtorMBT can be considered as continued work in progress while Boolector is

under active development. Our model-based testing workflow improved considerably since version 1.6 and our practical experience is extremely positive. Our experimental results confirm the claim in [5] that model-based API testing in particular in combination with delta debugging is effective for testing verification back-ends.

7.1 Workflow

The core test case generation engine in our model-based testing framework is our model-based API testing tool BtorMBT, which implements a model of Boolector's API and generates valid sequences of API calls. In case that one of these sequences causes an error, Boolector generates an API error trace, which allows to replay and reproduce faulty behaviour with our trace execution tool BtorUntrace. Our delta debugging tool ddMBT then minimizes such an API error trace while preserving its fault-inducing characteristics when replayed with BtorUntrace. Figure 7.1 describes the general workflow of our framework and its components as follows.

Data Model Boolector, our system under test, is an SMT solver for the quantifier-free theory of fixed-size bit-vectors, arrays, and uninterpreted functions as defined in the SMT-LIB v2 [13], and natively supports the use of non-recursive first-order lambda terms.

Option Model Boolector provides multiple solver engines, which are configurable via more than 70 options in total. All options and their default values and value ranges can be queried via its API. BtorMBT identifies valid option values based on these queries and defines invalid option combinations. Else, any random combination of options is allowed.

API Model Boolector provides a rich public API with full access to the complete feature set of the solver. It is available in C and Python, but since both BtorMBT and BtorUntrace, which tightly integrate Boolector via its API, are written in C, we will in the following focus on its C API, which consists of more than 150 API functions. Figure 7.2 illustrates the API model of Boolector as implemented in BtorMBT. It incorporates Boolector's option model and the data model as above, and will be described in more detail in Section 7.2.

BtorMBT Our model-based API testing tool BtorMBT generates test cases as valid sequences of calls to Boolector's API and implements the API model illustrated in Figure 7.2. It aims to exploit the full feature set of Boolector as available via its API. We will describe BtorMBT in more detail in Section 7.2.

API Error Trace Boolector provides the possibility to trace all API calls with their arguments into a dedicated trace file. Given such an API trace, the trace execution tool BtorUntrace then replays the sequence of API calls listed in the trace file and reproduces undesired behavior in case of an error. An example of an API trace generated by Boolector is given in Figure 7.3. We will describe API tracing and the trace in Figure 7.3 in Section 7.3 in more detail.

BtorUntrace The trace execution tool BtorUntrace allows to replay a sequence of API calls given an API trace file as above. We will describe BtorUntrace in Section 7.3 in more detail.

ddMBT Given an API Error trace file which logs a sequence of API calls leading to undesired behavior of Boolector, the delta debugger ddMBT minimizes the API trace while preserving this behavior reproduced via BtorUntrace. We will describe ddMBT in more detail in Section 7.4.

7.2 Test Case Generation with BtorMBT

Our model-based API tester BtorMBT is a dedicated tool for testing random configurations of Boolector. It is explicitly tailored to Boolector and supports the quantifier-free theories of fixed-size bit-vectors, arrays and uninterpreted functions, extended with non-recursive first-order lambda terms.

BtorMBT serves as the test case generation engine in our model-based testing workflow and fully supports all functionality provided by Boolector via its API. In contrast to input fuzzers such as FuzzSMT [30, 32], which generate a random but valid input file to be handed to the system under test, BtorMBT tightly integrates Boolector via its C API and generates test cases in the form of valid sequences of API calls. In case that an API sequence triggers an error, Boolector produces an API error trace, which can then be replayed with BtorUntrace for debugging purposes. BtorMBT further allows to test Boolector's cloning feature [91], which generates a disjunct copy of a Boolector instance, in a test setting we refer to as *shadow clone testing*. We will describe shadow clone testing in more detail in Section 7.2.2.

Note that Boolector makes heavy use of runtime assertions and provides means to internally check key features of the solver. This includes model validation for satisfiable instances, checking the inconsistency of the set of inconsistent assumptions (also called failed assumptions [55]) for unsatisfiable instances when incremental solving is enabled, and checks for Boolector's cloning feature. Errors triggered by BtorMBT therefore include failed internal checks, assertion failures, segmentation faults, and any other kind of abort.

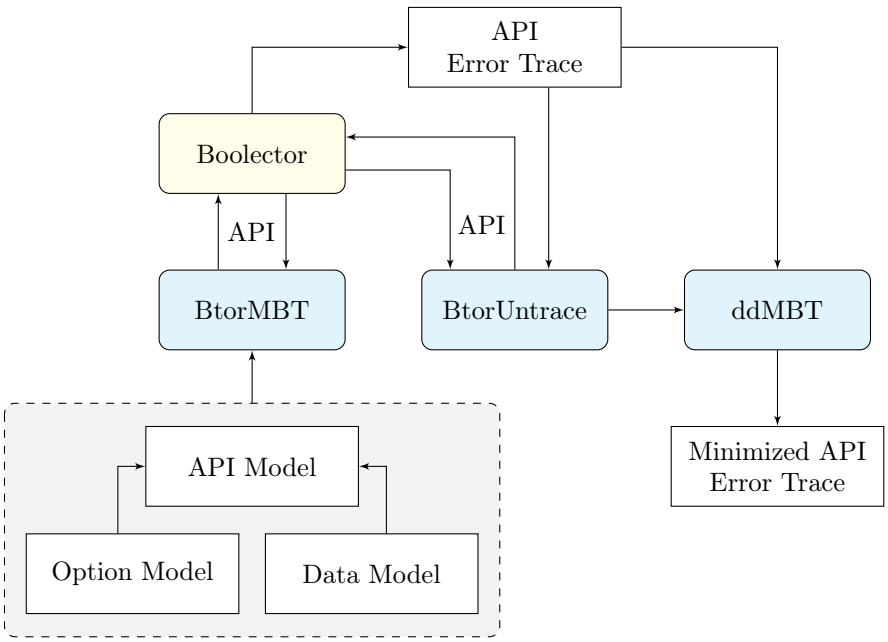


Figure 7.1: General workflow of model-based API testing for Boolector.

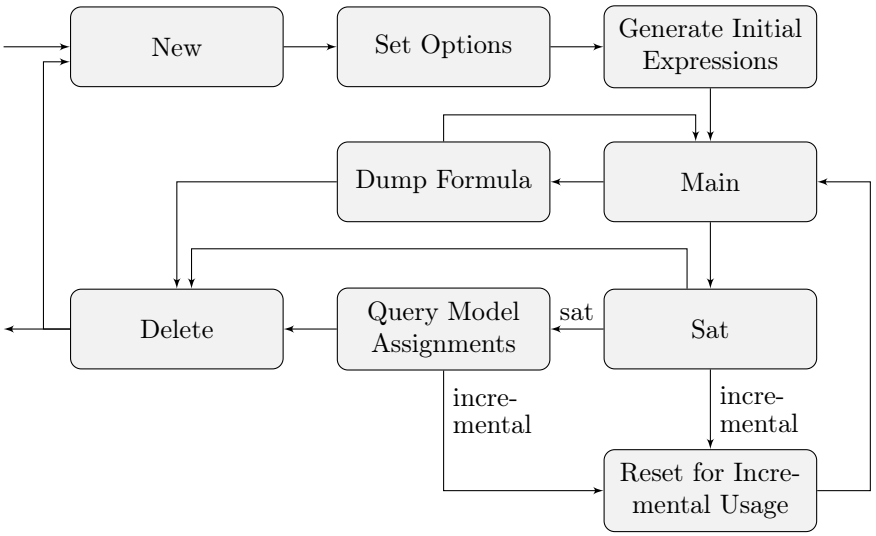


Figure 7.2: The API model of Boolector as implemented in BtorMBT.

7.2.1 Architecture

The general architecture of BtorMBT is defined by a state machine implementing the data, options and API Model of Boolector as illustrated in Figure 7.2. Test case generation with BtorMBT is performed in rounds, each with a different configuration of Boolector. One round corresponds to a sequence of states from state New to state Delete and the states are defined as follows.

New In each round, a fresh instance of Boolector is generated. Further, all parameters that influence formula size and structure such as probability distributions and maximum numbers for generating and releasing expressions are (re)initialized with random values (within certain ranges).

Set Options Boolector provides multiple solver engines, with some of them relying on an underlying SAT solver. As back-end SAT solver, Boolector supports the solvers Lingeling [23], PicoSAT [20] and MiniSat [55]. BtorMBT randomly chooses a solver engine and, if required, a SAT solver to use. The solver engine is then configured by randomly choosing and setting configuration options and their values within their predefined ranges. Note that option combinations identified as invalid according to the option model of Boolector are explicitly excluded. Further note that some options, e.g., incremental solving, are chosen with higher probability than others, depending on their relevance.

Generate Initial Expressions After a new Boolector configuration and all parameters that influence the formula size and structure have been set up, an initial set of inputs and non-input expressions is generated. The set of inputs is divided into randomly sized shares of uninterpreted functions, array variables, and Boolean and bit-vector constants and variables. Non-input expressions are randomly generated by combining inputs and already existing non-input expressions until the maximum number of non-input expressions is reached. Note that in case that the chosen solver engine only supports the quantifier-free theory of fixed-size bit-vectors, only bit-vector expressions are generated.

Main After generating an initial set of expressions, in state Main a random number of operations that influence the structure of the input formula is performed in random order: (1) new expressions are generated, (2) existing expressions are released, (3) and existing Boolean expressions are added to the input formula as assertions and, in the incremental case, assumptions. Note that when selecting expressions to generate new non-input expressions, in order to increase expression depth, expressions from the initial set are chosen with lower probability. After finalizing the current input formula, BtorMBT randomly performs various operations that operate on the current state of the input formula and

possibly manipulate the current state of the Boolector instance e.g., simplifying the input formula by means of rewriting and other techniques, or generating a clone of the current Boolector instance. Next, BtorMBT randomly picks between dumping the input formula (state Dump Formula) or determining its satisfiability (state Sat). Note that the latter is chosen with higher probability.

Dump Formula Boolector allows to dump the current state of the input formula (without assumptions) anytime during the solving process and supports BTOR [34], SMT-LIB v2 [16] and AIGER [19] as output format. Depending on the structure of the formula, either of these formats is chosen randomly. Note that AIGER is a bit-blasted and-inverter-graph (AIG) representation of the input formula and can therefore only be produced if it is a bit-vector formula without uninterpreted functions, arrays and lambda terms. If the output format is BTOR or SMT-LIB v2, the formula is dumped to a temporary file. This file is then parsed into a temporary Boolector instance in order to check the dump for errors. If the output format is AIGER, the formula is dumped to stdout without checking for correctness since Boolector does not provide support for parsing input files in AIGER format. Checking the correctness of AIGER dumps is left to future work. Next, BtorMBT randomly picks between concluding the current round (state Delete) and continuing (state Main) with equal probability.

Sat After setting up the current input, a call to determine its satisfiability is issued. Boolector supports incremental solving under assumptions and, in case of unsatisfiability, allows to determine the set of failed assumptions [55], i.e., those assumptions that are inconsistent with the input formula. If failed assumptions checking is enabled, the set of failed assumptions is internally checked for inconsistency with the current input. In case of satisfiability, Boolector provides a model of the input formula, and if model checking is enabled, this model is internally checked for validity. Next, if the input formula is satisfiable and model generation is enabled, BtorMBT continues with printing and querying model assignments (state Query Model Assignments). If incremental solving is enabled, BtorMBT may randomly choose to continue with an incremental step (state Reset for Incremental Usage). Else, it proceeds to conclude the current round (state Delete).

Query Model Assignments If the input formula is satisfiable and model generation is enabled, calls to query the model assignments of all generated expressions are issued. Further, if model printing is enabled, the model of the input formula is printed to stdout. Boolector supports model output formats based on BTOR [34] and SMT-LIB v2 [16], and BtorMBT may pick either of them randomly. Note that as of version 2.5, model output in SMT-LIB format is not yet fully standardized.

Reset for Incremental Usage Prior to performing an incremental step, parameters such as maximum numbers and probability distributions are reinitialized with random values (within certain ranges). Note that the value ranges for these parameters may differ from the ranges employed in state New.

Delete State Delete concludes one round (one test case) with releasing all generated expressions and deleting the current Boolector instance.

7.2.2 Shadow Clone Testing

As of version 2.0, Boolector provides a cloning feature which allows to generate a disjunct copy of a Boolector instance [91]. A clone captures the current state of the solver and can be either a deep copy (full clone) or a term layer copy (term layer clone) of the original instance. As a deep copy, a *full clone* includes the underlying SAT solver and the (bit-blasted) AIG layer (if present) and requires corresponding cloning support of the SAT solver back-end (e.g., Lingeling [22, 23]). A full clone is required to behave exactly the same as the instance it has been cloned from. Generating full clones for producing independent subproblems is, e.g., one of the key requirements for the work splitting approach implemented in PBoolector [98], a parallel prototype implementation of Boolector. A *term layer clone*, on the other hand, only copies the term layer of the original instance, which does not guarantee exact same behavior but is sufficient for many applications (e.g., generating a dual solver instance for the dual-propagation-based optimization of the lemmas on demand approach described in Paper C).

In order to test and guarantee that a full clone behaves *exactly* the same as the instance it has been cloned from, BtorMBT provides a dedicated *shadow clone* test setting similar to shadow clone testing as implemented for Lingeling. Shadow clone testing is randomly enabled and when enabled, BtorMBT initially generates a full clone (the shadow clone) of the current Boolector instance, which then mirrors every API call to the original instance and cross-checks return values for equivalence. Additionally, Boolector implements extensive checks for freshly generated clones and internally checks the state of the shadow clone after each API call. A shadow clone may be initialized anytime prior to the first SAT call and is usually randomly released and regenerated multiple times after being initialized, at different stages during one test round, to prevent that clones are only generated and checked prior to (incremental) API calls.

7.3 API Trace Execution with BtorUntrace

Our SMT solver Boolector allows to record all API calls with their arguments to a trace file, which then serves as input for our trace execution tool BtorUntrace. An example of an API trace generated by Boolector is given in Figure 7.3, with each line of the trace either listing an API call or the return value of an API call

```
1  new
2  return b1
3  set_opt b1 1 incremental 1
4  set_opt b1 14 rewrite-level 0
5  bitvec_sort b1
6  return s1@b1
7  array_sort b1 s1@b1 s1@b1
8  return s3
9  array b1 s3@b1 array1
10 return e2@b1
11 var b1 s1@b1 index1
12 return e3@b1
13 var b1 s1@b1 index2
14 return e4@b1
15 read b1 e2@b1 e3@b1
16 return e6@b1
17 read b1 e2@b1 e4@b1
18 return e8@b1
19 eq b1 e3@b1 e4@b1
20 return e9@b1
21 ne b1 e6@b1 e8@b1
22 return e-10@b1
23 assert b1 e9@b1
24 assume b1 e-10@b1
25 sat b1
26 return 20
27 failed b1 e-10@b1
28 return true
29 sat b1
30 return 10
31 release b1 e2@b1
32 release b1 e3@b1
33 release b1 e4@b1
34 release b1 e6@b1
35 release b1 e8@b1
36 release b1 e9@b1
37 release b1 e-10@b1
38 release_sort b1 s1@b1
39 release_sort b1 s3@b1
40 delete b1
```

Figure 7.3: An example API trace as generated by Boolector.

in chronological order. A line representing an API call consists of an identifier, the Boolector instance to issue the call to, and the arguments to the call. A line representing the return value of an API call must immediately follow the line of the call and consists of the keyword `return` and the return value, which can either be an identifier or a numerical value. As an example, consider the API call in line 7 and its return value in line 8. Identifier `array_sort` in line 7 refers to the API call to create an array sort with bit-vector sort s1 as its first (index sort) and second (element sort) argument, issued to Boolector instance b1. Line 8 identifies the return value of this call as array sort s3.

BtorUntrace is a dedicated tool for replaying traces generated by Boolector and tightly integrates Boolector via its C API. In our model-based API testing workflow, BtorUntrace is used in combination with BtorMBT to reproduce faulty behavior when a test case generated by BtorMBT fails. However, BtorUntrace is also useful outside of our testing workflow when debugging undesired behavior triggered by any (real world) application of Boolector. Since BtorUntrace only requires the API trace to replay a faulty run of Boolector, it is, e.g., not necessary to have the original (possibly complex) setup of the tool chain available for debugging purposes. Further, some errors triggered via the API may not be triggered with a dump of the corresponding input formula since some (sequences of) Boolector API calls can not be expressed in the input file formats it supports.

7.4 API Error Trace Minimization with ddMBT

Our delta debugger ddMBT minimizes a given API error trace while preserving its failure-inducing characteristics based on the exit code and error message produced by Boolector when replaying the trace with BtorUntrace. Trace minimization with ddMBT works in rounds until fixpoint, with each round divided into three phases. In the first phase, lines of the trace file are eliminated in a divide-and-conquer manner similar to the original delta debugging algorithm proposed in [70]. In the second and third phase, children of terms are substituted with fresh variables and already existing expressions of the same sort. These three substitution strategies are in practice usually sufficient to obtain a trace file small enough to allow efficient debugging. In some cases, however, modifying numeric parameters such as bit-widths, shift widths, or indices for slicing might be beneficial. We leave these enhancements to future work.

7.5 Experimental Evaluation

In the following, similarly as in [5], we evaluate the effectiveness of our model-based API tester BtorMBT in terms of code coverage, throughput and the success rate when inserting defects into the code of Boolector. We further compare the performance of BtorMBT to grammar-based input fuzz testing, in particular to FuzzSMT [30,32], the only currently available input fuzzer for SMT. Note that since FuzzSMT originally is an input fuzzer for the SMT-LIB v1 [97] language, we applied an available patch [103] for SMT-LIB v2 [13] support. However, this patch only provides SMT-LIB v2 compliant output of SMT-LIB v1 test cases. As a consequence, extensions of the language introduced in SMT-LIB v2, e.g., support for incremental solving, are not included, which may have a considerable impact on the performance of the tool. Measuring this impact without extending the tool to support the full feature set of the SMT-LIB v2 language, however, is difficult. We still include a comparison with FuzzSMT since up until now it was the de facto state-of-the-art for generating random test cases in SMT, and leave the extension of the tool to fully support SMT-LIB v2 to future work.

7.5.1 Configuration

Since we aim to evaluate BtorMBT and FuzzSMT on as even terms as possible, we provide a script for FuzzSMT that simulates option fuzzing and the round-based behavior as implemented in BtorMBT. In the following, we refer to this script as FuzzSMT, and compare against the version of BtorMBT released together with the current version 2.4 of Boolector.

We optionally switch off option fuzzing (while still randomly choosing solver engines and SAT solvers) and refer to BtorMBT and FuzzSMT without option fuzzing as BtorMBTno and FuzzSMTno. Note that we compiled Boolector with all three supported SAT solvers Lingeling [23], PicoSAT [20] and MiniSat [55].

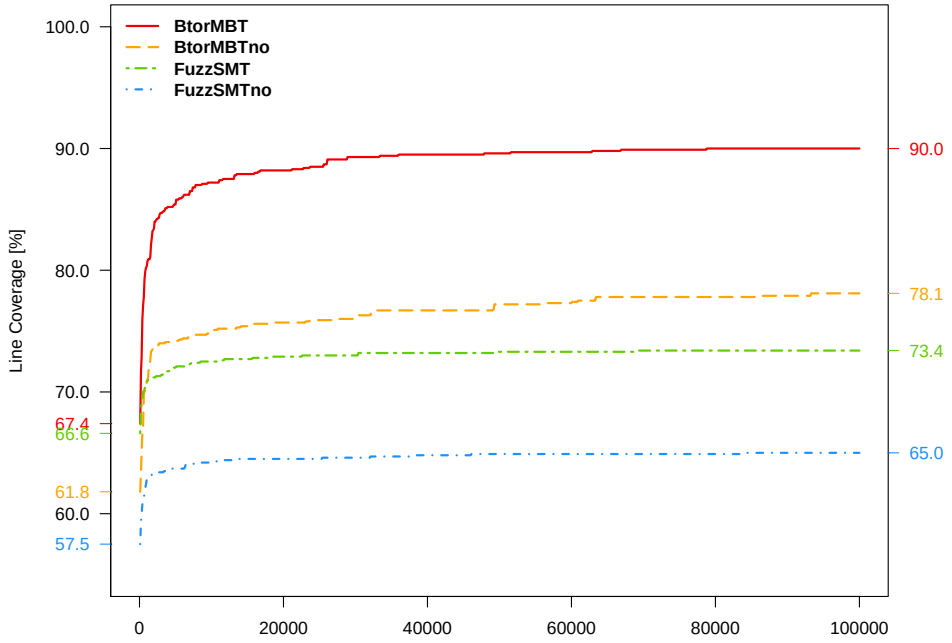


Figure 7.4: Code coverage evolution over 100k rounds.

We evaluated BtorMBT and FuzzSMT on runs of 100k rounds and chose a time limit of 2 seconds per round. Note that since increasing this time limit did not increase code coverage for 100k rounds for configuration BtorMBT, we chose this limit as a good compromise between throughput and test coverage.

In order to be able to determine if either of the tools is able to identify faulty mutations of Boolector (see Section 7.5.4), we thoroughly tested our base version 2.4 of Boolector prior to our experimental evaluation and run 10M rounds with each BtorMBT and FuzzSMT, all of which did not result in a single error.

We performed all our experiments on a cluster with 30 nodes of 2.83 GHz Intel Core 2 Quad machines running Ubuntu 14.04.5 LTS.

7.5.2 Code Coverage

We used the tool gcov of the GNU Compiler Collection (GCC) suite to determine the code coverage over 100k non-faulty rounds of BtorMBT and FuzzMBT with and without option fuzzing. The evolution of line coverage for configurations BtorMBT, BtorMBTno, FuzzSMT and FuzzSMTno as measured by gcov over 100k rounds is illustrated in Figure 7.4.

After 10k runs, BtorMBT already achieves 75% line coverage without option fuzzing, and 87% when option fuzzing is enabled. FuzzSMT, on the other hand, reaches a coverage of 64% and 72%. After 100k runs, BtorMBT improves coverage up to 78% (+3%) and 90% (+12%), while FuzzSMT achieves a coverage of 65% (+1%) and 73% (+1%).

Unsurprisingly, BtorMBT covers more than 98% of Boolector's API, with error handling code as the uncovered rest that is not triggered due to the fact that all test cases were error free. FuzzSMT, on the other hand, only achieves a coverage of 52%, which is likely to be improved by introducing full SMT-LIB v2 support, however, not up to the coverage rate BtorMBT achieved.

Note that for both tools, a coverage rate of 100% for error free test cases is in general impossible due to the fact that error handling code is not triggered. Further, since FuzzSMT only generates input in SMT-LIB v2 format, all code related to parsing BTOR format (3%) remains unused.

7.5.3 Throughput

When fuzz testing an SMT solver, no matter if it is input fuzzing or API fuzzing, the number of tests completed within a certain time frame (the throughput) is an important measure of efficiency and effectiveness of the test method. A high number may indicate that the generated test cases are too trivial and therefore less likely to trigger errors. A low number, on the other hand, may be caused by too difficult and therefore too time consuming test cases, which may considerably slow down progress when testing. We aim to perform as many good test cases, i.e., test cases with a high code coverage rate, in as little time as possible, which is a balancing act between the two extremes above.

For 100k rounds, BtorMBT achieves a throughput of on average 45 rounds per second, which increases by 20% when shadow clone testing is disabled. FuzzSMT, on the other hand, achieves a far lower throughput of 7 test cases per second since it first generates an input file that is then handed to the SMT solver. Further, FuzzSMT is written in Java, and (re)starting the Java VM in each round introduces additional overhead which further decreases throughput.

Note that in 100k rounds with BtorMBT, 20% of all calls to determine satisfiability are incremental. Further, one in four solved instances is satisfiable, which corresponds to a rather unbalanced ratio of 1:3 of satisfiable to unsatisfiable instances. We leave improving this ratio to future work.

7.5.4 Defect Injection

In our final experiment, we evaluated the success rate of BtorMBT and FuzzSMT in identifying faulty configurations of Boolector. We compiled a set of test configurations TC (4626 in total), which consists of two subsets TC_A and TC_D and contains configurations where we introduced artificial defects into the source code of Boolector. Set TC_A contains 2305 configurations with a randomly in-

		BtorMBT		**BtorMBTno**		**FuzzSMT**		**FuzzSMTno**	
		Found	[%]	Found	[%]	Found	[%]	Found	[%]
100k	TC_A (2305)	2088	90.6	1789	77.6				
	TC_D (2321)	1629	70.2	1366	58.9				
	TC (4626)	**3717**	**80.4**	**3155**	**68.2**				
10k	TC_A (2305)	2028	88.0	1719	74.6	1735	75.3	1523	66.1
	TC_D (2321)	1510	65.1	1277	55.0	1304	56.2	1153	49.7
	TC (4626)	**3538**	**76.5**	**2996**	**64.8**	**3039**	**65.7**	**2676**	**57.8**

Table 7.1: Number of faulty configurations of Boolector identified by BtorMBT, BtorMBTno, FuzzSMT and FuzzSMTno within 100k and 10k rounds.

serted abort statement, and set TC_D consists of 2321 configurations where we deleted a random statement from the code. All 4626 configurations in set TC are faulty configurations. However, some defects, e.g., modifications of heuristics due to a missing statement, may result in performance bugs rather than producing incorrect results or any other erroneous behavior and are therefore impossible to detect with either test method.

For each faulty test configuration, we set a limit of 100k rounds for BtorMBT. However, since the low throughput of FuzzSMT (7 rounds per second) would require too much runtime for our experiment with 100k rounds even on a cluster with 30 nodes (26 days in the worst case), we limited the number of rounds for FuzzSMT to 10k and compare its results to the number of faulty configurations identified by BtorMBT within 10k rounds.

Table 7.1 shows the number of faulty configurations identified by BtorMBT and FuzzSMT with and without option fuzzing within 100k and 10k rounds. Overall, within 10k rounds BtorMBT has a 11% higher success rate than FuzzSMT, which is increased by 14% to 80.4% when the limit is extended to 100k rounds. Disabling option fuzzing, on the other hand, decreases the number of configurations identified as faulty for both tools by 12%.

Not surprisingly, for both tools the success rates for configuration set TC_A correspond to their code coverage as determined in Section 7.5.2. The number of successfully identified faulty configurations in set TC_D, on the other hand, is significantly lower due to the fact that set TC_D contains test cases with defects that concern error handling code or decrease performance rather than introducing erroneous behavior. Further, in case of BtorMBT, since dumps in AIGER format are not tested for correctness it is not possible to detect configurations that produce incorrect AIGER output. The same applies in case of FuzzSMT for configurations that produce incorrect dump output in any format since it is not checked for correctness.

7.6 Discussion

Our model-based API fuzzer BtorMBT generates random but valid sequences of calls to Boolector's API and allows to test random configurations of Boolector on random input formulas. With a success rate of 80% on our artificial set of faulty configurations of Boolector and a line coverage of 90% over 100k rounds, our experiments suggest that BtorMBT is an effective method for testing Boolector. Our extremely positive practical experience confirms this claim.

Our model-based API testing framework is the core component of the testing workflow in the development process of Boolector. However, it still needs to be complemented by several other tools to cover cases that can not be tested with BtorMBT alone. The solver front end, for example, can only be tested by using the solver as standalone tool with files in BTOR or SMT-LIB format as input. For that purpose, we use a suite of regression tests and FuzzSMT, even though its support for the SMT-LIB v2 format is incomplete. Another example is parser testing, which is incomplete with BtorMBT since Boolector is currently not able to dump incremental input and in general does not use the full feature set of the SMT-LIB language when dumping. Further, dumping with Boolector only produces valid input files. However, a parser must be tested for correct parse error handling on invalid input, too. We test Boolector's parsers by means of the tool fzsexpr of the ddsexpr tool set [21], which generates (mostly) invalid input by mutating existing files based on lines, S-expressions and characters.

Currently, BtorMBT produces a rather unbalanced ratio of 1:3 of satisfiable to unsatisfiable instances. Further, AIGER dump output is not checked for correctness since Boolector does not allow to parse AIGER input files. We leave improving the ratio of satisfiable to unsatisfiable instances and checking the correctness of AIGER dumps to future work.

Chapter 8

Conclusion

In this thesis we presented several techniques for bit-precise reasoning in SMT that improve the current state-of-the-art and were the topic of four peer-reviewed publications, which are included as Papers A-D. We discussed the contributions and results of these publications and expanded on related topics of interest. Of all four publications, the author of this thesis is the main author.

In Paper A we first reimplemented the SLS for bit-vector logics approach in [58] in our SMT solver Boolector. We then improved the strategy in [58] with a propagation-based extension that takes full advantage of the word-level structure of the input formula by propagating assignments from the outputs to the inputs. Our results suggested that combining our propagation-based techniques with bit-blasting considerably improves performance. In Chapter 3 we implemented such a combination for our propagation-based strategy and the SLS for SMT approach in [58] and showed that combining local search techniques with a bit-blasting within a sequential portfolio setting indeed considerably improves performance. However, the metric used as a limit for our sequential portfolio combination with the SLS for SMT approach in [58] introduces too much overhead in terms of runtime. We leave finding a better metric to future work. In Chapter 3 we further provided an extensive experimental analysis of randomization effects for both the SLS for SMT approach in [58] (as implemented in Boolector) and our propagation-based techniques, which was not included in Paper A. Our results showed that our propagation-based techniques are more robust with respect to randomization effects than the SLS for SMT approach in [58].

The propagation-based strategy presented in Paper A still relies on brute-force randomization and restarts to achieve completeness. It further still has to fall back on regular SLS techniques as described in [58]. In Paper B we proposed an improvement of our propagation-based techniques and introduced a complete propagation-based local search procedure for quantifier-free bit-vector formulas in SMT. We further implemented a combination of our improved propagation-based techniques with bit-blasting, which considerably improved performance. However, we observed that on certain benchmarks, our propagation-based strategy is at a disadvantage compared to the SLS-based strategy proposed in [58]. In Chapter 4 we provided an in-depth analysis why this might be the case and observed that our propagation-based strategy struggles if the input formula contains a considerable amount of expressions with constant bits. Our results sug-

gest that introducing knowledge about constant bits into our propagation-based techniques will considerably improve performance. We leave this improvement to future work. Extending our techniques by introducing strategies for conflict detection and resolution during backtracing as well as lemma generation in order to obtain an algorithm that is able to also prove unsatisfiability is another challenging direction for future work.

In Paper C we proposed two optimization techniques for the LOD procedure for the quantifier-free theory combination of fixed-size bit-vectors, arrays and uninterpreted functions as implemented in Boolector. In Chapter 5 we then evaluated and analyzed the relation between our optimization techniques and a recent improvement of the base LOD procedure, which introduces a more eager lemma generation approach. We observed that generating lemmas more eagerly leaves less room for improvement for our optimization techniques. However, they still improve performance, in particular in terms of runtime. Further, generating lemmas more lazily significantly decreases the overhead introduced by the dual solver instance of our dual-propagation-based technique. Introducing techniques to render this overhead negligible is still left to future work.

We implemented all our techniques in our SMT solver Boolector, which contributed to winning several tracks of the SMT competitions 2014, 2015 and 2016. However, this would not have been possible without rigorous test methods to ensure correctness and robustness of the solver. In Paper D and Chapter 7 we addressed two such methods. In Paper D we introduced a delta debugging procedure for the SMT-LIB v2 language that enables solver developers to locate failure-inducing code in a time efficient manner. Our delta debugging tool is in particular efficient in combination with input fuzzing. State-of-the-art solvers such as Boolector, however, require additional test methods that are not entirely input-based but allow to exhaustively test the functionality provided by the solver via its API. In Chapter 7 we lifted the successful application of model-basted API fuzzing from SAT [5] to SMT and introduced our model-based API testing framework for Boolector. Overall, our results and practical experience suggest that our model-based API testing tool set is an effective method for testing Boolector. However, currently, our model-based API tester produces a rather unbalanced ratio of 1:3 of satisfiable to unsatisfiable instances and does not support checking the correctness of AIGER dumps. We leave improving this ratio and introducing correctness checks for AIGER dumps to future work.

Part II

Papers

A

Paper A

Improving Local Search for Bit-Vector Logics in SMT with Path Propagation

Published In Proceedings of the 4th International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS 2015), affiliated to the 15th International Conference on Formal Methods in Computer Aided Design (FMCAD 2015), 10 pages, Austin, TX, USA, 2015.

Authors Aina Niemetz, Mathias Preiner, Armin Biere and Andreas Fröhlich.

Fixes References to line numbers in procedure sat (Figure 1), return statement of procedure select_move (Figure 2), total number of benchmarks in our benchmark set in description of experimental setup, uniform number format for time columns in results tables, total time for configuration Z3sls in Table 1, number of benchmarks solved faster by configuration Bb+Bprop+frw (Figure 5, instances with a runtime < 0.01 seconds were previously not considered).

Abstract Bit-blasting is the main approach for solving word-level constraints in SAT Modulo Theories (SMT) for bit-vector logics. However, in practice it often reaches its limits, even if combined with sophisticated rewriting and simplification techniques. In this paper, we extended a recently proposed alternative based on stochastic local search (SLS) and improve neighbor selection based on down propagation of assignments. We further reimplemented the previous SLS approach in our SMT solver Boolector and confirm its effectiveness. We then added our novel propagation-based extension and provide an extensive experimental evaluation, which suggests that combining these techniques with Boolector's bit-blasting engine enables Boolector to solve substantially more instances.

1 Introduction

SAT Modulo Theories (SMT) procedures for deciding the satisfiability of first-order formulas w.r.t. the theory of fixed-size bit-vectors usually employ a so called *bit-blasting* approach, where the input formula is eagerly reduced to propositional logic (SAT). While efficient in practice, it heavily relies on rewriting to simplify the input prior to bit-blasting, and consequently, on the underlying SAT solver. This method, however, does in general not scale if the input size can not be reduced sufficiently. Lazy approaches based on DPLL(T) as in [38,67], on the other hand, aim to improve solver performance by employing a layered approach—in the latter case in a parallel portfolio setting together with the standard bit-blasting approach. Attacking the problem from a different angle, in [58], Fröhlich et al. proposed a stochastic local search (SLS) procedure to solve bit-vector formulas directly on the theory level, i.e., on the word-level, without the need for a SAT solver. In contrast to [66], where Griggio et al. attempted to reproduce previous successful applications of SLS in the SAT domain (e.g. [74]) by integrating a bit-level SLS solver with the SMT solver MathSAT [42], lifting SLS to the theory level delivered promising initial results. However, we argue that neighborhood exploration, as suggested in [58], does not yet fully exploit the advantage of working on the theory level. In essence, it mostly simulates bit-level local search by focusing on single bit flips.

In this paper, we first reimplemented the word-level local search approach introduced in [58] in our SMT solver Boolector, the winner of the QF_BV track of the SMT competition 2015, and confirm its effectiveness as presented in [58]. We then aim at improving neighbor selection as in [58] by introducing so called *propagation moves*. That is, rather than almost solely relying on bit flips of bit-vector and Boolean variables (driven by a scoring function), we introduce an additional strategy to satisfy lines by propagating assignments from the outputs to the inputs. We extended the SLS engine in Boolector with our propagation-based strategy and provide an extensive experimental evaluation which shows that using these techniques in combination with Boolector's bit-blasting engine in a sequential portfolio manner [104] considerably improves its performance.

2 SLS for QF_BV at a glance

The core SLS engine as implemented in Boolector is similar to the SLS architecture presented in [58] with some exceptions, which are mainly due to implementation issues. In the following, we will highlight all relevant differences and give an overview of the general workflow corresponding to the algorithm depicted in Figure 1. Note that, as in bit-level local search, the given word-level local search procedure is incomplete in the sense that it is not able to determine unsatisfiability.

Given a bit-vector formula ϕ, procedure *sat* initially applies several rewriting and simplification techniques to yield a simplified formula π (line 5), which in the following serves as the input to the actual SLS procedure. In contrast to [58], we do not impose restrictions to the bit-vector logic definition, i.e., we do not require π to be in Negation Normal Form (NNF). However, without loss of generality, we do restrict the set of Boolean expressions to the set of unary and binary operators $\{\neg, \wedge, =, <\}$. We further represent formula π as a directed acyclic graph (DAG) with (possibly) multiple roots, which we also refer to as *constraints*.

Given a set of constraints $\{a_1, ..., a_m\}$ in π, we adopt the constraint weighting scheme in [58] and associate each constraint a_i with a weight w_i, which is initialized with 1 and updated whenever no propagation and no regular SLS move could be found, and a random variable/value pair is chosen. As in [58], we then define the notion of states of a local search algorithm for an SMT bit-vector problem based on the values of the constraint weights and the assignments to its inputs, i.e., a set of Boolean and bit-vector variables. In the following, we refer to 0-arity bit-vector function symbols as *bit-vector variables*, and to numerical constants (e.g. #bvX in SMT-LIB notation [13]) as *bit-vector constants*. We further implicitly treat Boolean variables as bit-vector variables of bit-width one and include them in all definitions over bit-vector variables if not otherwise noted.

Given the simplified formula π, as in [58], we define the initial state of the SLS procedure by initializing all constraint weights with one (line 6) and all bit-vector variables with zero (line 8), and yield an initial assignment α. Starting from this initial assignment, the SLS procedure then iteratively moves to neighboring states until a satisfying assignment is found. The actual local search procedure consists of two loops (line 7-18), where the inner loop (line 10-18) represents a single round of search, and the outer loop realizes restarts after a certain number of moves has been performed. Given a constant c_2 (we choose $c_2 = 100$), the maximum number of moves in a single search round is defined as in [58] as

$$\text{max\_moves}\ (i) = \begin{cases} c_2 & \text{if } i \text{ is odd} \\ c_2 \cdot 2^{\frac{i}{2}} & \text{if } i \text{ is even.} \end{cases}$$

In each iteration of the outer loop, we compute the score for all Boolean expressions (line 9) as a floating value, and recursively define a scoring function s to drive the search and assess the quality of an assignment as follows.

Given α and a Boolean variable v, its score $s(v, \alpha)$ is defined as in [58] as its assignment in α, i.e.,

$$s(v, \alpha) = \alpha(v).$$

```
1  procedure sat (φ)
2      global α                                  // assignment
3      global s                                  // score
4      global π                                  // simplified formula
5      π := simplify (φ)
6      init_constraint_weights ()
7      for i = 1 to ∞
8          α := init_inputs (π)
9          s := compute_score ()
10         for j = 0 to max_moves (i)
11             if all_constraints_sat ()
12                 return SAT
13             root := select_constraint ()
14             (var, val) := select_move (root)
15             update_assignments (var, val)
16             if is_randomized_move (var, val)
17                 update_constraint_weights ()
18             update_score ()
```

Figure 1: The core SLS procedure in pseudo-code.

```
1  procedure select_move (root)
2      choose depending on ratio n:m
3          (var, val) := select_prop_move (root)
4      else
5          (var, val) := select_sls_move (root)
6      return (var, val)
```

Figure 2: Procedure *select_move* in pseudo-code. Move selection depends on a ratio $n : m$ of propagation to sls moves.

```
1  procedure select_sls_move (root)
2      V := select_vars (root)
3      choose with propability wp
4          (var, val) := random_walk (V)
5      else
6          (var, val) := find_best_move (V)
7      if (var, val) = none
8          (var, val) := randomize (V)
9      return (var, val)
```

Figure 3: A regular SLS move in pseudo-code.

Analogously, given α and the negation of a Boolean variable v, the score of $\neg v$ is defined as

$$s(\neg v, \alpha) = \neg\alpha(v).$$

Given two Boolean expressions a and b, the score of an and-expression over a and b is adopted from [58] as

$$s(a \wedge b, \alpha) = \tfrac{1}{2} \cdot (s(a, \alpha) + s(b, \alpha))$$
$$s(\neg(a \wedge b), \alpha) = max(s(\neg a, \alpha), s(\neg b, \alpha)).$$

The score of an equality over bit-vector expressions a and b of bit-width n is defined as in [58] via the hamming distance h of their assignments in α, i.e. given a constant $0 \leq c_1 \leq 1$ (we choose $c_1 = 0.5$), we define

$$s(a = b, \alpha) = \begin{cases} 1.0 & \text{if } \alpha(a) = \alpha(b) \\ c_1 \cdot \left(1 - \dfrac{h(\alpha(a),\, \alpha(b))}{n}\right) & \text{otherwise} \end{cases}$$

$$s(a \neq b, \alpha) = \begin{cases} 1.0 & \text{if } \alpha(a) \neq \alpha(b) \\ 0.0 & \text{otherwise.} \end{cases}$$

The score definition of an inequality over bit-vector expressions a and b of bit-width n significantly differs from [58] due to implementation issues and is defined via a function m, which is a cheap heuristic to determine an upper bound on the number of bit flips required such that $\alpha(a)$ and $\alpha(b)$ match the given inequality relation. Note that m is pessimistic in the sense that the actual number of required bit-flips might be smaller. The score of a bit-vector inequality is then defined as

$$s(a < b, \alpha) = \begin{cases} 1.0 & \text{if } \alpha(a) < \alpha(b) \\ c_1 \cdot \left(1 - \dfrac{m_<(\alpha(a), \alpha(b))}{n}\right) & \text{otherwise} \end{cases}$$

$$s(a \geq b, \alpha) = \begin{cases} 1.0 & \text{if } \alpha(a) \geq \alpha(b) \\ c_1 \cdot \left(1 - \dfrac{m_\geq(\alpha(a), \alpha(b))}{n}\right) & \text{otherwise.} \end{cases}$$

Lastly, given α, a set of constraints $\{a_1, \cdots, a_m\}$ in π, and its corresponding set of weights $\{w_1, \cdots, w_m\}$, we define the overall score of formula π as in [58] as $s(\pi, \alpha) = w_1 \cdot s(a_1, \alpha) + \cdots + w_m \cdot s(a_m, \alpha)$. Note that the score of a constraint a_i is normalized and therefore bound to $0 \leq s(a_i, \alpha) \leq 1$. The overall score of formula π, however, may be greater than 1.

As in [58], we perform score computation bottom-up, i.e., starting from the inputs. However, due to the fact that we do not require formula π to be in NNF, it is not possible to employ what [58] refers to as "early pruning". Hence, in order to still minimize the overhead for score computation, in contrast to [58] we do not recompute the score for all nodes in a formula's DAG representation on update. Rather, we identify the cone of influence, i.e., those parts of the formula affected by changing a given input, and update its score accordingly (line 18).

In each iteration of the inner loop (lines 10-18), depending on a yet unsatisfied constraint root and assignment α, a variable var and a value val is selected (line 14), and assignment α and all scores are updated accordingly (lines 15 and 18). Note that given constraints $\{a_1, \cdots, a_m\}$ in π, constraint root is selected as in [58] as the unsatisfied constraint a_i that, given a constant c_3 (we choose $c_3 = 20$), maximizes

$$s(a_i, \alpha) + c_3 \cdot \sqrt{\frac{\log selected(a_i)}{nmoves}},$$

where $selected(a_i)$ is the number of times a_i has already been selected, and $nmoves$ is the overall number of moves performed so far.

In the general SLS case, i.e., without enabling our additional propagation strategy, we adopt the notion of (extended) neighborhood for regular SLS moves from [58], which includes single bit flips, increment, decrement, and bitwise negation. Move selection is then performed as in [58] corresponding to Figure 3 as follows. Given an unsatisfied constraint root, all bit-vector variables reachable while traversing from the root to the inputs are collected into a set of candidate variables V (line 2). Out of all possible combinations in V and its extended neighborhood, the variable/neighbor pair with the most improvement of the overall score $s(\pi, \alpha)$ is determined as the best move (line 6). If no best move is found, a random variable and value is chosen (line 8), and the weights of all constraints are updated as in [58] (Figure 1, line 17). That is, with a probability sp (we choose $sp = 0.95$), the weights of all unsatisfied constraints are increased by 1. Otherwise, the weights of all satisfied constraints are decreased to a minimum of 1.

Note that in the general SLS case, all moves performed are regular SLS moves as described in Figure 3, which corresponds to a ratio $0 : \infty$ of propagation moves to regular SLS moves in Figure 2. Further, as in [58], *select_sls_move* optionally supports so called *random walks*, i.e., if enabled, with a probability wp (we choose $wp = 0.1$) a random move out of all variable / neighbor combinations is chosen.

3 Propagation Moves

The notion of (extended) neighborhood in [58] combines a simple SAT-style SLS neighborhood relation, given by flipping single bits of a variable assignment, with

three additional bit-vector moves: increment, decrement, and bitwise negation. With a neighborhood size of $n + 3$ for a single bit-vector variable of bit-width n, it is easy to see that exploring its neighborhood is dominated by bitwise flips. As a consequence, neighborhood exploration in [58] mainly simulates bit-level local search without fully exploiting possible benefits of the word-level. Further, for variables with increasing bit-width, SLS moves as described in Section 2 become increasingly expensive.

Even though the approach in [58] showed promising results on the *Sage2* benchmark family, it still struggles in general in comparison to state-of-the-art bit-blasting approaches. Especially when dealing with a certain type of problem, the shortcomings of the chosen neighborhood relations become evident. Consider the following example in SMT-LIB notation:

```
(set-logic QF_BV)
(declare-fun v () (_ BitVec 65))
(assert (= (_ bv18446744073709551617 65) (bvmul (_ bv274177 65) v)))
(check-sat)
(exit)
```

Assuming that we disable possible simplification techniques, it is not possible to determine the (single) solution $v = 67280421310721$ within a time limit of 1200s on a 3.4GHz Intel Core i7-2600 machine (355837 moves, 21 restarts) with the SLS procedure as described in Section 2. With our propagation-based strategy, however, the example above is solved instantly within one single propagation move. In this section, we will introduce this strategy and its application in detail as follows.

3.1 Propagation-Based Move Selection

When enabling our propagation-based strategy, within the core procedure as described in Figure 1, we support three different scenarios, depending on the type of move to be selected (line 14):

(i) all moves performed are propagation moves
(i.e., propagation and regular SLS moves are performed with a ratio $\infty : 0$)

(ii) propagation and regular SLS moves are performed with a ratio $n : m$

In case of propagation moves, move selection is performed corresponding to Figure 4 as follows. Given an unsatisfied constraint root, a simplified formula π, and assignment α, we force root to be true (line 3) and iteratively propagate its new assignment along one path towards the inputs while assuming all other paths to be fixed with respect to assignment α.

```
1  procedure select_prop_move (root)
2      cur := root
3      val := 1
4      loop
5          if is_bv_var (cur)
6              return cur, val
7          if not has_non_const_input (cur)       // conflict
8              return select_sls_move (root)      // recover with SLS move
9          // path selection
10         if is_ite (cur)
11             if flip_cond ()
12                 cur := get_cond_node (cur)
13                 val := flip_bv (α(cur))
14             else
15                 cur := get_enabled_branch (cur)
16             continue
17         if is_boolean_and (cur) and has_exactly_one_ctrl_input (cur)
18             inp := get_ctrl_input (cur)
19         else
20             inp := select_random_input (cur)
21         // path propagation
22         oth := get_other_inputs (cur, inp)
23         val := compute_value (cur, val, oth)
24         if val = none                          // conflict
25             return select_sls_move (root)      // recover with SLS move
26         cur := inp
```

Figure 4: A propagation move in pseudo-code.

In each iteration, path selection (lines 10-20) is implemented as choosing one of the current node's inputs, which in general happens randomly except for two cases. If the current node is an if-then-else (ite) node (line 10), with probability ip (we choose $ip = 0.1$), the first (and else, the second) out of two options is chosen:

(1) flip the condition, or

(2) assume the assignment of the condition to be fixed and follow the enabled branch.

In (1), we reset cur to the condition and val to the flipped value of its assignment in α (lines 12-13). In (2), we reset cur to the enabled branch (line 15) before continuing with the next iteration.

If the current node is a Boolean AND node (line 17), on the other hand, path selection is based on the notion of (a posteriori) observability don't cares as defined in the context of ATPG [87]. Given an unsatisfied constraint root, while propagating its flipped assignment $\alpha(\text{root}) = 1$ along one path towards the inputs, as a consequence, the assignments of all AND-gates along this path are flipped as well. Given a concrete assignment to the inputs of an AND-gate, however, we can determine lines that do not influence its output under the current assignment. Consequently, if the output of an AND node is currently assigned to 0 (i.e., to be flipped to 1), we follow its controlling input, i.e., the input with controlling value 0, if only one of its inputs is controlling. Else, we choose randomly.

Note that the procedure in Figure 4 aims at finding an assignment for a bit-vector variable. As a consequence, during path selection, we do not choose bit-vector constants as input inp. If a node cur has only bit-vector constants as inputs, assignment val conflicts with the inputs of cur and we recover with a regular SLS move (line 8).

After selecting a path, we implement down propagation of assignments by computing a new assignment for input inp as the inverse of the current node given its assignment and the assignment to its other inputs, which are assumed to be fixed (line 23). Note that in general, if the other inputs are not bit-vector constants and such an inverse value does not exist, procedure compute_value concludes with an assignment for the chosen input that matches the assignment of the current node, disregarding its other inputs. However, if its other inputs are bit-vector constants, it concludes with 'none' and we again recover with a regular SLS move (line 25). Finally, if we were able to successfully propagate all assignments along a path from root to a bit-vector variable, we conclude with this variable and its new assignment (line 5).

Note that procedure *select_prop_move* optionally supports to force a random walk rather than performing a regular SLS move (where random walks, if enabled, occur only with probability wp, otherwise) when recovering from a conflicting assignment (lines 8 and 25).

3.2 Propagating Assignments via Inverse Computation

Down propagation of assignments is implemented by procedure compute_value via computing the inverse of a given node (representing a bit-vector operation) given its assignment and the assignment of all but one of its inputs. Without loss of generality, we restrict the set of bit-vector operations to {=, bvnot, bvult, bvshl, bvshr, bvadd, bvand, bvmul, bvudiv, bvurem, concat, extract} as defined in the SMT-LIB standard v2 [13], where all but bvnot and extract are binary operations. Note that for some operations in this set no well-defined inverse operation exists. In that case, procedure compute_value in general produces non-unique values via randomization (of bits or bit-vectors). Further note that given the assignment of the bit-vector operation and one of its inputs, if the input is not a bit-vector constant and no inverse value could be found, compute_value disregards the assignment of the given input and chooses an inverse value that matches the assignment of the given operation. However, if the input is a bit-vector constant, its assignment conflicts with the assignment of the given operation, and compute_value concludes with 'none'.

In the following, we assume that given a bit-vector v, its least significant bit (LSB) is positioned at index 0 (LSB = $v[0]$). We further denote a bit-vector slice expression (extract in SMT-LIB notation) from index i to j (incl.) with $v[i:j]$ and assume that i and j are numerical constants. Procedure compute_value determines the assignment x of an input of a given operation, given its assignment c and the assignment a of its other input (if any), as follows:

$=$

Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{a} = \mathbf{x}$ or $\mathbf{c} \coloneqq \mathbf{x} = \mathbf{a}$, its inverse with respect to x is defined as $x \coloneqq a$ if $c = 1$, and a random value other than a, otherwise.

bvnot

Given a bit-vector expression $\mathbf{c} \coloneqq \sim \mathbf{x}$, its inverse with respect to x is defined as $x \coloneqq \sim c$.

bvult

- Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{a} < \mathbf{x}$, we determine its inverse with respect to x as follows:

  (1) If $c = 1$ and $a \neq 2^n - 1$, we choose a random value for x with $a < x$.

  (2) If $c = 0$, we choose a random value for x with $a \geq x$.

  (3) If a is not a bit-vector constant, and neither 1) nor 2) apply, we disregard a and choose a random value for x with $x > 0$.

  If a is a bit-vector constant and neither 1) nor 2) apply, the current value of c is conflicting with a and we can not find a value for x.

- Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{x} < \mathbf{a}$, we determine its inverse with respect to x as follows:

  (1) If $c = 1$ and $a \neq 0$, we choose a random value for x with $x < a$.

  (2) If $c = 0$, we choose a random value for x with $x \geq a$.

  (3) If a is not a bit-vector constant, and neither 1) nor 2) apply, we disregard a and choose a random value for x with $x < 2^n - 1$.

  If a is a bit-vector constant and neither 1) nor 2) apply, the current value of c is conflicting with a and we can not find a value for x.

bvshl

- Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{a} << \mathbf{x}$, its inverse with respect to x is defined as the number n_0 of least significant bits set to 0 in c. If a is a bit-vector constant and n_0 is equal to the bit-width of a, or if $a << n_0$ and c do not match, the current value of c is conflicting with a and we can not find a value for x.

- Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{x} << \mathbf{a}$, its inverse with respect to x is defined as $x \coloneqq c >> a$. Note that the bits shifted in may be set arbitrarily. If a is a bit-vector constant and the a least significant bits in c are not set to 0, the current value of c is conflicting with a and we can not find a value for x.

bvshr

Is defined analogous to bvshl.

bvadd

Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{a} + \mathbf{x}$ or $\mathbf{c} \coloneqq \mathbf{x} + \mathbf{a}$, its inverse with respect to x is defined as $x \coloneqq c - a$.

bvand

Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{a}$ & $\mathbf{x}$ or $\mathbf{c} \coloneqq \mathbf{x}$ & $\mathbf{a}$, its inverse with respect to x is defined depending on the bits set in both a and c, i.e., given position i,

- if $c[i] = 1$, then $x[i] \coloneqq 1$
- if $c[i] = 0$ and $a[i] = 1$, then $x[i] \coloneqq 0$
- if $c[i] = 0$ and $a[i] = 0$, then $x[i] \coloneqq dc$

Note that don't care values (dc) may be set arbitrarily. If a is a bit-vector constant, and $a[i] = 0$ for any position i where $c[i] = 1$, the current value of c is conflicting with a and we can not find a value for x.

bvmul

Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{a} \cdot \mathbf{x}$ or $\mathbf{c} \coloneqq \mathbf{x} \cdot \mathbf{a}$ of bit-width n, we determine its inverse with respect to x as follows.

(1) If a is a divisor of c, then $x \coloneqq c/a$.

(2) If $gcd(a, 2^n) = 1$, we determine the multiplicative inverse a^{-1} of value a via the Extended Euclidean algorithm and define $x \coloneqq a^{-1} \cdot c$.

(3) If neither applies and a is not a bit-vector constant, we try if any $v \in \{7, 5, 3, 2\}$ is a divisor of c. If so, we choose $x \coloneqq c \,/\, v$, and $x \coloneqq 1$, otherwise.

If a is a bit-vector constant and neither 1) nor 2) apply, the current value of c is conflicting with a and we can not find a value for x.

bvudiv

- Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{a}/\mathbf{x}$ of bit-width n, we determine its inverse with respect to x as follows:

  (1) If $a = c = 0$, we choose a random value for x.

  (2) If $a \neq 0$, and c is a divisor of a, then $x \coloneqq a/c$.

  (3) If a is not a bit-vector constant, and neither 1) nor 2) apply, we disregard a, and choose a random value x such that $x \cdot c$ does not overflow.

  If a is a bit-vector constant and neither 1) nor 2) apply, the current value of c is conflicting with a and we can not find a value for x.

- Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{x}/\mathbf{a}$ of bit-width n, we determine its inverse with respect to x as follows:

  (1) If $a = 0$ and $c = 2^n - 1$, we choose a random value for x. This is due to the fact, that given a value v of bit-width n, Boolector handles division by zero as $v/0 = 2^n - 1$.

  (2) If $a \neq 0$, and $c \cdot a$ does not overflow, then $x \coloneqq c \cdot a$.

  (3) If a is not a bit-vector constant, and neither 1) nor 2) apply, we disregard a, choose a random value v such that $v \cdot c$ does not overflow, and define $x \coloneqq c \cdot v$.

  If a is a bit-vector constant and neither 1) nor 2) apply, the current value of c is conflicting with a and we can not find a value for x.

bvurem

- Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{a} \mod \mathbf{x}$ of bit-width n, we determine its inverse with respect to x as follows:

  (1) If $a = c$ and $a \neq 2^n - 1$, we choose a random value for x with $x > a$.

  (2) If $a > c$ and $a - c > c$, since $c + m \cdot x = a$ we choose $m = 1$ and define $x \coloneqq a - c$.

  (3) If a is not a bit-vector constant, and either $a > c$ and $a - c \leq c$, or $a < c$ and $c < 2^n - 1$, we disregard a and choose a random value for x with $x > c$.

  If $a = c = 2^n - 1$, or if a is a bit-vector constant and 2) does not apply, the current value of c is conflicting with a and we can not find a value for x.

- Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{x} \mod \mathbf{a}$ of bit-width n, we determine its inverse with respect to x as follows:

  (1) If $a > c$, with probability 0.5 we either choose

  a) $x \coloneqq c$, or,

  b) since $c + m \cdot a = x$ we choose m such that $c + m \cdot a$ does not overflow and define $x \coloneqq c + m \cdot a$. If $c + a$ overflows, but a is not a bit-vector constant, we choose a).

  (2) If $a \leq c$ and a is not a bit-vector constant, we choose $x \coloneqq c$.

  If a is a bit-vector constant and 1) does not apply, current value of c is conflicting with a and we can not find a value for x.

concat

- Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{a} \circ \mathbf{x}$ where a is of bit-width m and c of bit-width n, its inverse with respect to x is defined as the slice $x \coloneqq c[0 : n - m - 1]$. If a is a bit-vector constant and $c[n - m : n - 1] \neq a$, the current value of c is conflicting with a and we can not find a value for x.

- Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{x} \circ \mathbf{a}$ where a is of bit-width m and c of bit-width n, its inverse with respect to x is defined as the slice $x \coloneqq c[m : n - 1]$. If a is a bit-vector constant and $c[0 : m - 1] \neq a$, the current value of c is conflicting with a and we can not find a value for x.

extract

Given a bit-vector expression $\mathbf{c} \coloneqq \mathbf{x}[\mathbf{l} : \mathbf{u}]$, where x is of bit-width n (and c of bit-width $u - l + 1$), we determine its inverse with respect to x given a position i as follows. If $l \geq i \leq u$, then $x[i] = c[i - l]$. Else, we choose a random value for $x[i]$.

4 Experiments

We implemented the core SLS engine and its propagation-based extension in our SMT solver Boolector as described in Sections 2 and 3, and provide an evaluation of the following configurations.

(1) **Bb** The core Boolector engine, which uses a bit-blasting approach. This configuration is a version close to the version that won the QF_BV track of the SMT competition 2015.

(2) **Bsls** The SLS Boolector engine, optionally with random walks enabled (+rw).

(3) **Bprop** The SLS Boolector engine with our propagation-based strategy enabled. This configuration by default uses propagation moves only. It optionally supports the configuration of a ratio $n : m$ of propagation to regular SLS moves (+n:m) and conflict recovery via random walk rather than performing a regular SLS move (+frw).

We further compare our Boolector configurations Bsls(+rw) against the original implementation of [58] in Z3 [49] and refer to version 4.4.0 of Z3 with its SLS engine enabled as configuration Z3sls. Note that Z3sls enables random walks by default.

We compiled a benchmark set from all benchmarks with status *sat* and *unknown* in the QF_BV category of the SMT-LIB [14] benchmark library, and excluded all benchmarks that configuration Bb proved to be unsatisfiable within a time limit of 1200 seconds (16436 instances in total). We excluded 449 benchmarks from the *Sage2* benchmark family that were not SMT-LIB v2 compliant due to non-compliant operators.

All experiments were performed on a cluster with 30 nodes of 2.83GHz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 14.04.2 LTS. The results in [58] indicate that even though there still exists a considerable gap between the performance of state-of-the-art bit-blasting and word-level local search as introduced in [58], the latter significantly outperforms bit-blasting on several instances. Based on these findings, we evaluated our Bsls and Bprop configurations with regard to an application within a sequential portfolio setting and set a time limit of 10 seconds for all solver instances of these and the Z3sls configurations. The memory limit for each solver instance was set to 7GB. In case of a time or memory out, a penalty of the given time limit was added to the total CPU time.

Table 1 summarizes the results of configurations Bb, Bsls, Bsls+rw, and Z3sls on our benchmark set, grouped by family, within a time limit of 10 seconds. Configuration Bsls+rw corresponds to the configuration of Z3sls as both enable random walks, whereas configuration Bsls disables random walks by default. On

	Bb		Bsls		Bsls+rw		Z3sls	
Family	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
asp (376)	**46**	**3542.7**	0	3760.0	0	3760.0	0	3760.0
bench (223)	223	0.1	**223**	**0.0**	**223**	**0.0**	**223**	**0.0**
bmc (22)	**21**	**56.8**	11	118.1	12	114.6	7	150.5
brubiere2 (10)	3	71.9	10	4.3	**10**	**3.1**	1	90.2
brubiere3 (6)	4	45.6	6	4.0	**6**	**0.2**	2	40.0
brubiere4 (10)	0	100.0	9	10.0	9	10.0	**10**	**0.0**
calypto (13)	4	92.6	**4**	**91.6**	3	100.1	3	100.2
check2 (1)	**1**	**0.0**	1	0.0	1	0.0	0	10.0
crafted (1)	1	0.0	1	0.0	1	0.0	1	0.0
dwp (103)	103	0.3	**103**	**0.0**	**103**	**0.0**	103	0.9
fft (19)	**4**	**160.5**	0	190.0	0	190.0	0	190.0
float (126)	**23**	**1094.7**	2	1253.0	0	1260.0	0	1260.0
gulwani (6)	**5**	**17.3**	1	50.3	1	51.8	1	50.0
mcm (155)	**14**	**1452.4**	8	1508.0	5	1517.0	8	1473.5
pspace (21)	0	210.0	**21**	**16.3**	**21**	**16.3**	0	210.0
rubik (3)	**2**	**17.1**	0	30.0	0	30.0	0	30.0
RWS (20)	**13**	**97.8**	0	200.0	0	200.0	0	200.0
sage (6236)	**6236**	**2179.2**	5315	10497.0	5275	10626.9	5969	5261.0
Sage2 (6981)	1037	63022.8	604	64306.7	620	64289.8	**1597**	**56890.2**
spear (1675)	**1524**	**7434.7**	1188	6348.9	1187	6336.6	456	13288.6
stp (1)	0	10.0	0	10.0	0	10.0	0	10.0
stp_s (149)	**149**	**3.6**	127	410.0	128	421.3	149	6.1
tacas07 (3)	2	20.2	2	10.2	2	10.2	**2**	**10.1**
uclid (262)	258	426.4	29	2533.2	23	2551.1	**259**	**297.4**
VS3 (10)	0	100.0	0	100.0	0	100.0	0	100.0
wienand (4)	0	40.0	0	40.0	0	40.0	0	40.0
total (16436)	**9673**	**80196.8**	7665	91491.6	7630	91635.0	8791	83468.9

Table 1: Results by benchmark family for configurations Bb, Bsls, Bsls+rw, and Z3sls with a time limit of 10 seconds.

family *pspace*, Z3sls terminated with an error, for which a penalty of the given time limit was added to the total CPU time. Compared to the bit-blasting configuration Bb, as in [58], the overall results of configuration Bsls+rw confirm both the effectiveness of the SLS approach on certain instances, and the overall gap in performance. The performance of configurations Bsls+rw and Z3sls, however, considerably differs on some benchmark families. Z3sls outperforms Bsls+rw by 694 and 977 instances on the *sage* and *Sage2* benchmarks, whereas on the *spear* family, with a difference of 731 instances, it is vice versa. Running Bsls+rw with different seeds for initializing the random number generator, however, has almost no influence on the number of solved instances. Out of 10 randomly seeded runs of configuration Bsls+rw on our benchmark set, we observed a maximum deviation of 0.04% in the number of solved instances. We therefore believe that the difference in performance between Bsls+rw and Z3sls is mainly due to the following reasons. Similar to the influence of rewriting and simplification techniques on the performance of state-of-the-art bit-blasting approaches, rewriting and the choice of rewriting and simplification techniques considerably influence the performance of the actual SLS procedure. Given our benchmark set and a time limit of 10 seconds, Boolector's SLS engine in configuration Bsls+rw with rewriting and simplification disabled solves 2166 less instances. The choice of rewriting and simplification techniques employed by Boolector and Z3 differs significantly, which might be one reason for the difference in performance on certain benchmark families. Further, even though Bsls+rw corresponds to Z3sls as far as implementation issues allowed, both still differ in several implementation aspects, in particular, the scoring function for the $<$ operator, which might influence the performance of the SLS approach considerably.

The overall results in Table 1 imply that enabling random walks in Boolector's SLS engine does not improve its performance. We therefore use configuration Bsls as base for our further experiments. Further note that even though the last two authors of this paper were also involved in [58], the reimplementation of the approach in [58] within Boolector by the first author should be considered as independent. The results of Table 1, as a consequence, should be considered as an independent confirmation of the results in [58].

Table 2 compares the results of our base SLS configuration Bsls to configuration Bprop (propagation moves only), and Bprop+100:1, Bprop+10:1, Bprop+1:1, Bprop+1:10, and Bprop+1:100 (with ratios 100:1, 10:1, 1:1, 1:10, and 1:100 of propagation moves to regular SLS moves) on our benchmark set, grouped by family, within a time limit of 10 seconds. Overall, the results suggest that configurations with a higher ratio of propagation to regular SLS moves perform better in terms of solved instances and runtime. With an additional 90 solved instances, configuration Bprop shows the most improvement in comparison to configuration Bsls. Out of the 7755 instances solved by Bprop, more than 56% (4366 instances) were solved with propagation moves only, i.e., no recovery by means of a regular SLS move was necessary; and for roughly 85% (6568

Family	Bsls		Bprop+1:100		Bprop+1:10		Bprop+1:1		Bprop+10:1		Bprop+100:1		Bprop	
	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
asp (376)	0	3760.0	0	3760.0	0	3760.0	0	3760.0	0	3760.0	0	3760.0	0	3760.0
bench (223)	**223**	**0.0**	223	0.1	223	0.1	**223**	**0.0**	**223**	**0.0**	**223**	**0.0**	**223**	**0.0**
bmc (22)	11	118.1	12	116.5	**12**	**109.5**	12	117.9	10	123.2	10	121.9	11	118.3
brubiere2 (10)	10	4.3	10	4.5	10	4.2	**10**	**3.2**	10	4.1	10	4.2	10	4.2
brubiere3 (6)	6	4.0	6	18.6	**6**	**3.8**	6	12.9	6	13.2	6	9.5	6	11.9
brubiere4 (10)	9	10.0	9	10.0	9	10.0	**10**	**5.1**	10	7.7	10	7.7	10	7.7
calypto (13)	4	92.6	4	96.2	3	100.1	**4**	**90.2**	**4**	**90.2**	3	100.1	3	100.1
check2 (1)	1	0.0	1	0.0	1	0.0	1	0.0	1	0.0	1	0.0	1	0.0
crafted (1)	1	0.0	1	0.0	1	0.0	1	0.0	1	0.0	1	0.0	1	0.0
dwp (103)	103	0.0	103	0.0	103	0.0	103	0.0	103	0.0	103	0.0	103	0.0
fft (19)	0	190.0	0	190.0	0	190.0	**1**	**183.0**	0	190.0	0	190.0	0	190.0
float (126)	2	1253.0	3	1246.0	0	1260.0	3	1247.9	7	1216.6	7	1209.2	**8**	**1205.7**
gulwani (6)	**1**	**50.3**	**1**	**50.3**	1	50.8	1	51.5	1	58.4	0	60.0	0	60.0
mcm (155)	8	1508.0	8	1508.0	7	1505.8	**9**	**1500.3**	6	1513.4	5	1519.2	5	1518.8
pspace (21)	21	16.3	21	16.4	**21**	**16.2**	0	210.0	0	210.0	21	93.9	0	210.0
rubik (3)	0	30.0	0	30.0	0	30.0	0	30.0	0	30.0	0	30.0	0	30.0
RWS (20)	0	200.0	0	200.0	0	200.0	0	200.0	0	200.0	0	200.0	0	200.0
sage (6236)	**5315**	**10497.0**	5278	10604.1	5256	10868.5	5217	11330.1	5097	11925.0	5088	11983.7	5133	11827.0
Sage2 (6981)	604	64307.7	626	64211.7	**635**	**64205.3**	571	64613.7	615	64456.3	589	64536.5	606	64420.3
spear (1675)	1188	6348.9	1188	6339.9	1189	6324.3	1316	5190.2	1482	3087.0	1483	3039.3	**1485**	**3029.2**
stp (1)	0	10.0	0	10.0	0	10.0	0	10.0	0	10.0	0	10.0	0	10.0
stp_s (149)	127	410.0	127	410.2	130	407.0	**132**	**365.8**	124	412.7	127	398.1	127	399.8
tacas07 (3)	2	10.2	2	10.2	2	10.2	2	10.2	2	10.2	2	10.2	2	10.2
uclid (262)	**29**	**2533.2**	28	2534.7	13	2553.4	1	2618.1	9	2557.2	17	2559.5	21	2555.3
VS3 (10)	0	100.0	0	100.0	0	100.0	0	100.0	0	100.0	0	100.0	0	100.0
wienand (4)	0	40.0	0	40.0	0	40.0	0	40.0	0	40.0	0	40.0	0	40.0
total (16436)	7665	91490.6	7651	91505.2	7622	91759.0	7623	91690.4	7711	90015.3	7706	89982.9	**7755**	**89808.5**

Table 2: Results by benchmark family for configurations Bsls, Bprop (propagation moves only), and Bprop+n:m (with ratio 1:100, 1:10, 1:1, 10:1, and 100:1 of propagation moves to regular SLS moves) with a time limit of 10 seconds.

instances), less than 8 recovery moves were required. When recovering with a random walk rather than a regular SLS move, i.e., in configuration Bprop+frw, out of 7540 solved instances roughly 72% (5427 instances) required less than 8 recovery moves. This suggests that regular SLS recovery moves, even though expensive, yield a better performance. Given a time limit of 1 second, though, as depicted in Table 3, configuration Bprop+frw outperforms Bprop by 85 instances. In particular on benchmark family *spear*, within 1 second and compared to configurations Bb, Bsls, and Bprop, configuration Bprop+frw solves 1586, 843, and 363 more instances. These results suggest a combination of configurations Bb and Bprop+frw into configuration Bb+Bprop+frw in a sequential portfolio manner [104], where configuration Bprop+frw is run prior to bit-blasting for a given time limit of 1 second. In the following, configuration Bb+Bprop+frw is a virtual configuration as it has not been realized within Boolector yet. All results attributed to Bb+Bprop+frw have been determined based on the results of configuration Bb and Bprop+frw on our benchmark set with a time limit of 1200 and 1 seconds, respectively, as follows. On instances where configuration Bprop+frw timed out within 1 second, a penalty of 1 second has been added to the result of configuration Bb, else the result of configuration Bprop+frw was chosen.

Table 4 summarizes the results of configurations Bb and Bb+Bprop+frw on our benchmark set, grouped by family, within a time limit of 1200 seconds. Even though invoking the Bprop+frw engine prior to bit-blasting produces an overhead of 1 second for instances that Bprop+frw can not solve within the given time limit, the combination of both engines considerably improves the performance of the bit-blasting approach both in terms of solved instances and total runtime. Overall, configuration Bb+Bprop+frw is able to solve 38 more instance than configuration Bb, while requiring only 96% of its total runtime. In particular benchmark families *brubiere4*, *Sage2*, and *spear* show the most improvement with speed-ups of up to a factor of 39. Even when considering the 1 second overhead of Bb+Bprop+frw on each benchmark of the full QF_BV benchmark set of the SMT-LIB, which is a total of 50429 instances including all unsat benchmarks, the performance gain of 85181 seconds on the benchmarks of our set of 16436 instances still yields a performance improvement in terms of the overall runtime.

The results in Table 4 are further illustrated in Figure 5 by means of a scatter plot. It shows that configuration Bb+Bprop+frw solves benchmarks up to orders of magnitude faster than the pure bit-blasting approach Bb. In fact, 156, 595, and 1464 instances are solved more than 1000, 100, and 10 times faster when combining Bprop+frw with Bb. The overhead generated by unsuccessful Bprop+frw invocations, i.e., for instances that cannot be solved by Bprop+frw within the 1 second time limit, does not outweigh the performance gain by the successful ones.

	Bsls		Bprop		Bprop+frw	
Family	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
asp (376)	0	376.0	0	376.0	0	376.0
bench (223)	**223**	**0.0**	**223**	**0.0**	223	0.2
bmc (22)	10	12.9	**10**	**12.8**	9	13.6
brubiere2 (10)	**9**	**2.7**	9	3.3	3	7.0
brubiere3 (6)	4	2.0	4	3.7	**6**	**2.1**
brubiere4 (10)	**9**	**1.0**	8	2.0	8	2.0
calypto (13)	3	10.0	3	10.1	**5**	**9.0**
check2 (1)	1	0.0	1	0.0	1	0.0
crafted (1)	1	0.0	1	0.0	1	0.0
dwp (103)	103	0.0	103	0.0	103	0.0
fft (19)	0	19.0	0	19.0	0	19.0
float (126)	0	126.0	**3**	**124.3**	2	124.7
gulwani (6)	**1**	**5.3**	0	6.0	0	6.0
mcm (155)	1	154.7	**2**	**154.3**	1	154.7
pspace (21)	**20**	**16.3**	0	21.0	0	21.0
rubik (3)	0	3.0	0	3.0	0	3.0
RWS (20)	0	20.0	0	20.0	0	20.0
sage (6236)	**5038**	**1385.2**	4971	1418.4	4769	1579.1
Sage2 (6981)	**509**	**6604.9**	473	6657.1	452	6614.0
spear (1675)	819	1249.4	1299	604.2	**1662**	**186.6**
stp (1)	0	1.0	0	1.0	0	1.0
stp_s (149)	**74**	**94.4**	73	93.9	23	127.4
tacas07 (3)	2	1.2	2	1.2	2	1.2
uclid (262)	0	262.0	0	262.0	0	262.0
VS3 (10)	0	10.0	0	10.0	0	10.0
wienand (4)	0	4.0	0	4.0	0	4.0
total (16436)	6827	10361.1	7185	9807.4	**7270**	**9543.7**

Table 3: Results by benchmark family for configurations Bsls, Bprop, and Bprop+frw with a time limit of 1 second.

	Bb		Bb+Bprop+frw	
Family	Solved	Time [s]	Solved	Time [s]
asp (376)	**285**	**147790.3**	285	148075.3
bench (223)	**223**	**0.1**	223	0.2
bmc (22)	**22**	**58.8**	22	71.5
brubiere2 (10)	**10**	**842.8**	10	847.9
brubiere3 (6)	6	49.4	**6**	**2.1**
brubiere4 (10)	1	10980.7	**8**	**2400.0**
calypto (13)	7	9307.5	**7**	**8358.5**
check2 (1)	1	0.0	1	0.0
crafted (1)	1	0.0	1	0.0
dwp (103)	103	3.3	**103**	**0.0**
fft (19)	**5**	**17014.5**	5	17019.5
float (126)	**94**	**53264.1**	94	53356.1
gulwani (6)	**6**	**34.2**	6	40.2
mcm (155)	**56**	**135932.1**	56	135983.0
pspace (21)	**21**	**687.9**	21	708.9
rubik (3)	**3**	**94.5**	3	97.5
RWS (20)	**16**	**5031.7**	16	5047.7
sage (6236)	**6236**	**2179.2**	6236	3526.9
Sage2 (6981)	5621	2213659.7	**5648**	**2148247.0**
spear (1675)	1671	12747.3	**1675**	**323.4**
stp (1)	1	15.5	1	16.5
stp_s (149)	**149**	**3.6**	149	130.7
tacas07 (3)	3	28.7	**3**	**19.7**
uclid (262)	**262**	**434.1**	262	696.1
VS3 (10)	**3**	**8844.0**	3	8847.0
wienand (4)	0	4800.0	0	4800.0
total (16436)	14806	2623801.1	**14844**	**2538615.8**

Table 4: Results by benchmark family for configurations Bb and Bb+Bprop+frw with a time limit of 1200 seconds.

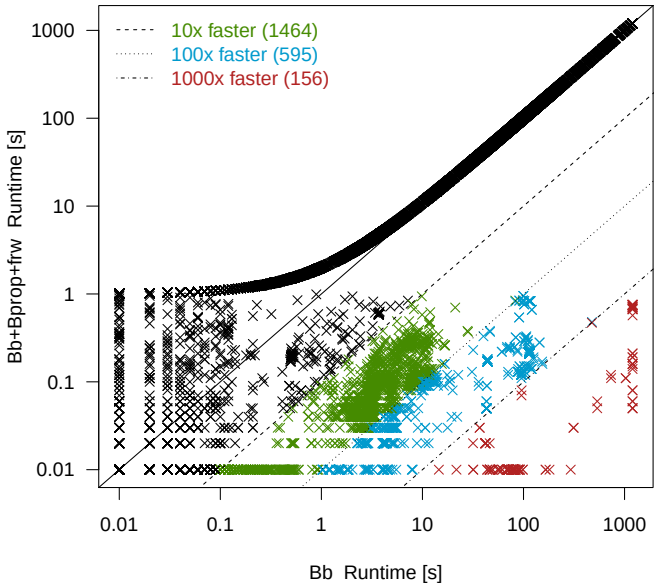


Figure 5: Bb vs. Bb+Bprop+frw on our benchmark set with a time limit of 1200 seconds and a 1 second time limit for Bprop+frw.

Finally, we tried to further increase the time limit for Bprop+frw of configuration Bb+Bprop+frw to 2 and 3 seconds. Compared to the 1 second time limit, with a time limit of 2 seconds, Bb+Bprop+frw solves an additional 8 instances while requiring 3500 seconds less runtime. The best results, however, are achieved with a time limit of 3 seconds, where Bb+Bprop+frw solves an additional 14 instances requiring 3700 seconds less runtime compared to the 1 second time limit.

5 Conclusion

In this paper, we reimplemented the word-level local search procedure presented in [58] within our SMT solver Boolector and independently confirmed its effectiveness. We further introduced an extension based on propagating assignments from the outputs to the inputs and evaluated our approach in particular with respect to a combination with Boolector's bit-blasting engine, which considerably improves its performance on satisfiable instances.

We chose all parametric values either as in [58], or such that they provide a good overall performance if newly introduced. Further optimizing those values with respect to performance is left to future work.

Our technique currently still relies on regular SLS tactics in certain conflict scenarios. However, this might not be necessary if path propagation via inverse computation guarantees that search space is not inadvertently pruned on conflict. This might currently be the case due to short cuts introduced for some bit-vector operations when e.g. choosing the simplest (and not some randomized) valid solution. Extending inverse computation to eliminate this kind of short cuts, and the implementation of the sequential portfolio style combination of our techniques and Boolector's state-of-the-art bit-blasting engine is left to future work.

Binaries of Boolector and all log files of our experimental evaluation can be found at `http://fmv.jku.at/difts15-prop`.

B

Paper B

Propagation Based Local Search for Bit-Precise Reasoning

Accepted For the Special Issue on Recent Topics in Satisfiability Modulo Theories of the International Journal on Formal Methods in System Design (FMSD), to appear. This Paper is an extended version of [92], published in Part I of the Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016), pages 179–186, Toronto, ON, Canada, 2016. It is included as a preprint version and may slightly differ from the final version.

Authors Aina Niemetz, Mathias Preiner and Armin Biere.

Abstract Many applications of computer-aided verification require bit-precise reasoning as provided by Satisfiability Modulo Theories (SMT) solvers for the theory of quantifier-free fixed-size bit-vectors. The current state-of-the-art in solving bit-vector formulas in SMT relies on bit-blasting, where a given formula is eagerly translated into propositional logic (SAT) and handed to an underlying SAT solver. Bit-blasting is efficient in practice, but may not scale if the input size can not be reduced sufficiently during preprocessing. A recent approach lifting stochastic local search (SLS) from the bit-level (SAT) to the word-level (SMT) without bit-blasting proved to be quite effective on hard satisfiable instances, particularly in the context of symbolic execution. However, it still relies on brute-force randomization and restarts to achieve completeness. Guided by a completeness proof, we simplified, extended and formalized our propagation-based variant of the SLS for SMT approach. We obtained a clean, simple and more precise algorithm that does not rely on SLS techniques, brute-force randomization or restarts to achieve completeness and yields substantial gain in performance. In this article, we discuss our complete propagation based local search approach for bit-vector logics in SMT in detail. We further provide an extended and extensive experimental evaluation including an analysis of randomization effects.

1 Introduction

A majority of applications in the field of hardware and software verification requires bit-precise reasoning as provided by Satisfiability Modulo Theories (SMT) solvers for the quantifier-free theory of fixed-size bit-vectors. In many of these applications, e.g., (constrained random) test case generation [88, 101, 105] or white box fuzz testing [62], a majority of the problems is satisfiable. For this kind of problems, local search procedures are useful even though they do not allow to determine unsatisfiability. Previous work [58, 93] showed that local search approaches for bit-vector logics in SMT are orthogonal to other approaches, which suggests that they are in particular beneficial within a portfolio setting [93].

Current state-of-the-art SMT solvers for the quantifier-free theory of fixed-size bit-vectors [11, 42, 49, 53, 91] employ the so-called bit-blasting approach (e.g., [80]), where an input formula is eagerly translated to propositional logic (SAT) and handed to an underlying SAT solver. While efficient in practice, bit-blasting approaches heavily rely on rewriting and other techniques [29, 37, 38, 39, 57, 59, 60, 67, 68] to simplify the input during preprocessing and may not scale if the input size can not be reduced sufficiently. In [58], Fröhlich et al. proposed to attack the problem from a different angle and lifted stochastic local search (SLS) from the bit-level (SAT) to the word-level (SMT) without bit-blasting. In previous years, and in particular since the SAT challenge 2012 [7], a new generation of SLS for SAT solvers with very simple architecture [8] achieved remarkable results not only in the random but also in the combinatorial tracks of recent SAT competitions [6, 7, 18]. However, previous attempts to utilize SLS techniques in SMT by integrating an SLS SAT solver into the DPLL(T)-framework of the SMT solver MathSAT [42] were not able to compete with bit-blasting [66]. In contrast, the SLS for SMT approach in [58] showed promising initial results. However, [58] does not fully exploit the word-level structure but rather simulates bit-level local search by focusing on single bit flips. Hence, in [93], we proposed a propagation-based extension of [58], which introduced an additional strategy to propagate assignments from the outputs to the inputs. This significantly improved performance. Our results further suggested that these techniques may be beneficial in a sequential portfolio setting [104] in combination with bit-blasting. However, [93] still relies on brute-force randomization and restarts to achieve completeness, so does [58]. Further, focusing only on inverse values as in [93] when down-propagating assignments may inadvertently prune the search.

In this paper, guided by a formal completeness proof we present a simple, precise and complete local search variant of the procedure proposed in [93]. Our approach does not use SLS techniques but relies on propagation of assignments only. It further does not require brute-force randomization or restarts to achieve completeness. To determine propagation paths, we extend the concept of controlling inputs to the word-level, which allows to further prune the search. To propagate assignments down, we lift the Automatic Test Pattern Generation (ATPG) [82] concept of "backtracing", which goes back to the PODEM algo-

rithm [63], to the word-level. We further provide a formalization of backtracing for the bit-level and the word-level. Note that in contrast to backtracing in ATPG, our algorithm works with complete assignments. Existing algorithms for word-level ATPG [73, 75] are based on branch-bound, use neither backtracing nor complete assignments, and in general lack formal treatment.

We implemented our techniques in our SMT solver Boolector and show that combining our propagation-based strategy with bit-blasting within a sequential portfolio setting is beneficial in terms of performance. We provide an extensive experimental evaluation, including an analysis of randomization effects as a result of different seeds for the random number generator, in particular in comparison to the SLS for SMT approach in [58] as implemented in Boolector. Our results show that our techniques yield a substantial gain in performance.

This article extends and revises work presented earlier in [92]. We provide a more detailed description of the propagation-based local search strategy introduced in [92], including extensive examples illustrating the core concepts of our approach. We further include a complete set of rules for determining assignments during backtracing. Our previous experimental evaluation of a sequential portfolio combination of our propagation-based technique with bit-blasting was a virtual experiment. For this paper, we implemented such a sequential portfolio combination within Boolector and provide an extensive experimental evaluation of our techniques. This evaluation includes the evaluation of randomization effects of our techniques, which was not included in previous work.

2 Overview

Our propagation-based local search procedure is based on propagating target assignments from the outputs to the inputs and does not need to rely on restarts or brute-force randomization to achieve completeness. Our notion of completeness follows the traditional notion of non-deterministic computation of Turing machines and is equivalent to the more established property of *probabilistically approximately complete* (PAC) [71], which is commonly used in the AI community to discuss completeness properties of local search algorithms. Note that this entails that we treat probabilistic choices as non-deterministic choices [71].

The basic idea of our approach is illustrated in Figure 1 and described more precisely in pseudo code in Figure 2. It is applied to propositional formulas (the bit-level) and quantifier-free bit-vector formulas (the word-level) as follows.

Given a formula ϕ, we assume without loss of generality that ϕ is a directed acyclic graph (DAG) with a single root r (the so-called root constraint or output of ϕ). We use the letter σ to refer to complete but non-satisfying assignments to all inputs and operators in ϕ. We further identify complete satisfying assignments with the letter ω. Starting from a random but non-satisfying initial assignment σ_1 with $\sigma_1(r) = 0$, our goal is to reach a satisfying assignment ω with $\omega(r) = 1$ by iteratively changing the values of primary inputs. We identify

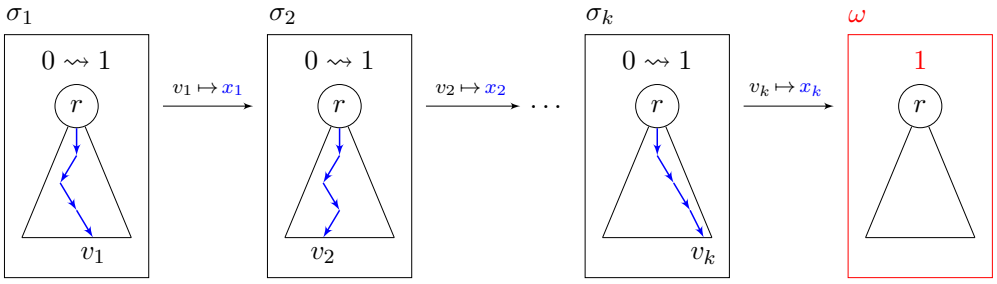


Figure 1: Basic idea of our propagation-based (local search) strategy. Starting from an unsatisfying assignment σ_1, we force root constraint r to assume its target value $\omega(r) = 1$ and iteratively propagate this information towards the primary inputs until we reach a satisfying assignment ω.

$\omega(r) = 1$ as the *target* value of output r (line 3), denoted as $0 \rightsquigarrow 1$ in Figure 1, and propagate this value along a path towards the primary inputs (lines 4-7. We also refer to this process as "backtracing" [63].

Recursively propagating target value $\omega(r) = 1$ from the output to the primary inputs yields a new value $x_i \neq \sigma_i(v_i)$ for an input v_i (e.g., x_1 for v_1 in Figure 1). By updating assignment σ_i on input v_i to $\sigma_{i+1}(v_i) = x_i$ (e.g., $\sigma_2(v_1) = x_1$ in Figure 1) without changing the value of other primary inputs but recomputing consistent values for inner nodes (lines 8-9), we move from σ_i to σ_{i+1} and repeat this process until we reach a satisfying assignment, i.e., $\sigma_{i+1} = \omega$.

When down-propagating assignments, we identify path selection (line 5) and selecting the value to propagate (line 6) as the only sources of non-determinism. However, we aim to maximally reduce non-deterministic choices without sacrificing completeness. Hence, on the bit-level, path selection prioritizes controlling inputs w.r.t. the current assignment, a well-known concept from ATPG, while value selection for a selected input is uniquely defined. On the word-level, we introduce the corresponding new notion of essential inputs, which lifts the bit-level concept to the word-level, and restrict value selection to the computation of what we refer to as consistent and inverse values.

As expected for local search, our propagation-based approach is not able to determine unsatisfiability. Thus the algorithm in Figure 2 does not terminate in case that a given input formula is unsatisfiable. When determining satisfiability, however, our strategy is complete (PAC), i.e., there exists a non-deterministic choice of moves according to the strategy that leads to a solution.

In the following, we first introduce and formalize our propagation-based approach on the bit-level and prove its completeness. We then lift it to the word-level, and prove its completeness on the word-level. We further analyze randomization effects as result of using different seeds for the random number generator and show that our techniques yield substantial performance improvements, in particular in combination with bit-blasting within a sequential portfolio setting.

```
1  function sat (r, σ)
2      while σ(r) ≠ 1                  // while not satisfied
3          g ≔ r, t ≔ 1                // initialize path as root path
4          while ¬leaf (g)             // while current node is an operator
5              n ≔ child (σ, t, g)     // select backtracing node
6              x ≔ value (σ, t, g, n)  // select backtracing value
7              g ≔ n, t ≔ x            // backtracing step (propagation)
8          if ¬constant (g)            // check if leaf is variable v = g
9              σ ≔ update (σ, g, t)    // apply move to variable v = g
10     return true                     // return with true if satisfied
```

Figure 2: The core sat procedure in pseudo-code.

3 Bit-Level

For the sake of simplicity and without loss of generality we consider a fixed Boolean formula ϕ and restrict the set of Boolean operators to $\{\wedge, \neg\}$. We interpret ϕ as a single-rooted And-Inverter-Graph (AIG) [81], where an AIG is a DAG represented as a 5-tuple (r, N, G, V, E).

The set of *nodes* $N = G \cup V$ contains the single root node $r \in N$, and is further partitioned into a set of *gates* G and a set of *primary inputs* (or *variables*) V. We require that the set of variables is non-empty, i.e., $V \neq \emptyset$, and assume that the Boolean constants $\mathbb{B} = \{0, 1\}$, i.e., $\{\mathit{false}, \mathit{true}\}$, do not occur in N. This assumption is without loss of generality since every occurrence of *true* and *false* as input to a gate $g \in G$ can be eliminated through rewriting.

The set of *gates* $G = A \cup I$ consists of a set of *and*-gates A and a set of *inverter*-gates I. We write $g = n \wedge m$ if $g \in A$, and $g = \neg n$ if $g \in I$. We further refer to the children of a node $g \in G$ as its (gate) inputs (e.g., n or m). Let $E = E_A \cup E_I$ be the *edge* relation between nodes, with $E_A \colon A \to N^2$ and $E_I \colon I \to N$ describing edges from *and*- resp. *inverter*-gates to its input(s). We write $E(g) = (n, m)$ for $g = n \wedge m$ and $E(g) = n$ for $g = \neg n$ and further introduce the notation $g \to n$ for an edge between a gate g and one of its inputs n.

We define a *complete assignment* σ of the given fixed formula ϕ as a *complete* function $\sigma \colon N \to \mathbb{B}$. Similarly, a *partial assignment* α of formula ϕ is defined as a *partial* function $\alpha \colon N \to \mathbb{B}$. We say that a complete assignment σ is *consistent* on a gate $g \in I$ with $g = \neg n$ iff $\sigma(g) = \neg\sigma(n)$, and *consistent* on a gate $g \in A$ with $g = n \wedge m$ iff $\sigma(g) = \sigma(n) \wedge \sigma(m)$.

A complete assignment σ is *globally consistent* on ϕ (or just *consistent*) iff it is consistent on all gates $g \in G$. An assignment σ is *satisfying* if it is consistent (thus complete) and satisfies the root, i.e., $\sigma(r) = 1$. We use the letter ω to denote a satisfying assignment. A formula ϕ is satisfiable if it has a satisfying assignment. We use $\mathcal{C}$ to denote the set of *consistent* assignments, and $\mathcal{W}$ with

$\mathcal{W} \subseteq \mathcal{C}$ to denote the set of *satisfying* assignments of formula ϕ.

Given two consistent assignments σ and σ', we say that σ' is obtained from σ by *flipping* the (assignment of a) variable $v \in V$, written as $\sigma \xrightarrow{v} \sigma'$, iff $\sigma(v) = \neg\sigma'(v)$ and $\sigma(u) = \sigma'(u)$ for all $u \in V \backslash \{v\}$. We also refer to flipping a variable as a *move*. Note that $\sigma'(g)$ for gates $g \in G$ is defined implicitly due to consistency of assignment σ' after fixing the values for the primary inputs V.

Given a set of variables V that can be flipped non-deterministically, let $S \colon \mathcal{C} \to \mathbb{P}(\mathcal{M})$ be a (local search) *strategy* that maps a consistent assignment to a set of possible moves $\mathcal{M} = V$. A move $v \in V$ is *valid* under strategy S for assignment $\sigma \in \mathcal{C}$ if $v \in S(\sigma)$. Similarly, a sequence of moves $\mu = (v_1, \ldots, v_k) \in V^*$ of length $k = |\mu|$ with $v_1, \ldots, v_k \in V$ is *valid* under strategy S, iff there exists a sequence of consistent assignments $(\sigma_1, \ldots, \sigma_{k+1}) \in \mathcal{C}^*$ such that $\sigma_i \xrightarrow{v_i} \sigma_{i+1}$ and $v_i \in S(\sigma_i)$ for $1 \leq i \leq k$. In this case, assignment σ_{k+1} can be *reached* from assignment σ_1 under strategy S (with k moves), also written as $\sigma_1 \to^* \sigma_{k+1}$.

Definition 1 (Complete Strategy)**.** If formula ϕ is satisfiable, then a strategy S is called *complete* iff for all consistent assignments $\sigma \in \mathcal{C}$ there exists a satisfying assignment $\omega \in \mathcal{W}$ such that ω can be reached from σ under S, i.e., $\sigma \to^* \omega$.

Given an assignment $\sigma \in \mathcal{C}$ and a satisfiable assignment $\omega \in \mathcal{W}$, let $\Delta(\sigma, \omega) = \{v \in V \mid \sigma(v) \neq \omega(v)\}$ be the set of variables with different values in σ and ω. Thus, $|\Delta(\sigma, \omega)|$ is the *Hamming Distance* between σ and ω on V.

Definition 2 (Distance-Reducing Strategy)**.** A strategy S is (non-deterministically) *distance reducing*, if for all assignments $\sigma \in \mathcal{C} \backslash \mathcal{W}$ there exists a satisfying assignment $\omega \in \mathcal{W}$ and a move $\sigma \xrightarrow{v} \sigma'$ valid under S which reduces the Hamming Distance. That is, move $v \in \mathcal{M}$ is in $\Delta(\sigma, \omega)$, thus $|\Delta(\sigma, \omega)| - |\Delta(\sigma', \omega)| = 1$.

Obviously, any distance reducing strategy can reach a satisfying assignment (though not necessarily ω) within at most $|\Delta(\sigma, \omega)|$ moves. This first observation is the key argument in the completeness proofs for our propagation based strategies (both on the bit-level and word-level).

Proposition 3. *A distance reducing strategy is also complete.*

In the following, our ultimate goal is to define a strategy that maximally reduces non-deterministic choices without sacrificing completeness. In the algorithm shown in Figure 2, path selection (selecting the backtracing node in line 5) and value selection (selecting the backtracing value in line 6) while down-propagating assignments constitute the only sources of non-determinism. As we will show later, in contrast to value selection on the word-level, selecting a backtracing value on the bit-level is uniquely defined. When selecting a backtracing node on the bit-level, non-determinism can be reduced by utilizing the notion of *controlling inputs* from ATPG [82], which is defined as follows.

Definition 4 (Controlling Input)**.** Let node $n \in N$ be an input of a gate $g \in G$, i.e., $g \to n$, and let σ be a complete assignment consistent on g. We say that

input n is *controlling* under σ if for all complete assignments σ' consistent on g with $\sigma(n) = \sigma'(n)$ we have $\sigma(g) = \sigma'(g)$.

In other words, gate input n is *controlling* if the assignment of g (i.e., its output value) remains unchanged as long as the assignment of n does not change.

Given an assignment σ consistent on $g \in G$, we denote a *target value* t for gate g as $\sigma(g) \rightsquigarrow t$. On the bit-level, target value t is *implicitly* given through assignment σ as $t = \neg\sigma(g)$, i.e., t can not be reached as long as the controlling inputs of g remain unchanged. On the word-level, t may be any value $t \neq \sigma(g)$.

Example 5. Figure 3 shows all possible assignments σ consistent on a gate $g \in G$. At the outputs we denote current assignment $\sigma(g)$ and target value t as $\sigma(g) \rightsquigarrow t$ with $t = \neg\sigma(g)$, e.g., $0 \rightsquigarrow 1$. At the inputs we show their assignment under σ. All controlling inputs are indicated with an underline. Note that *and*-gate $g = n \wedge m$ has *no* controlling inputs for $\sigma(g) = 1$.

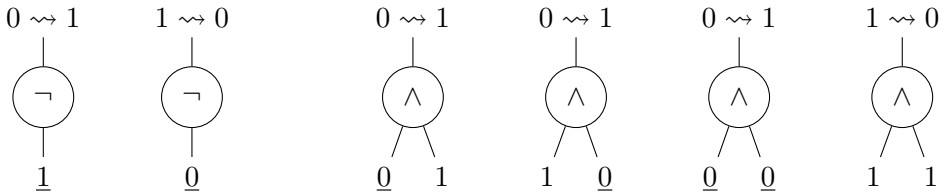


Figure 3: An *inverter*-gate and an *and*-gate and their controlling (underlined) inputs. Given output values indicate the transition from current to target value.

We define a sequence of nodes $\pi = (n_1, \ldots, n_k) \in N^+$ as a *path* of length k with $k = |\pi|$ iff $n_i \to n_{i+1}$ for $0 < i < k$, also written as $n_1 \to \ldots \to n_k$. A path π is *rooted* if $n_1 = r$, and (fully) *expanded* if $n_k \in V$. We refer to $n_k \in V$ as the *leaf* of π in this case. As a restriction on ϕ, we require all nodes $n \in N$ to be reachable from the root r, i.e., there exists a path π such that $\pi = (r, \ldots, n)$. We further require all paths to be acyclic, i.e., for all $n \in N$ there exists no path $n \to^+ n$. Note that as a consequence of representing ϕ as a DAG, this is the case for any path in ϕ. Given a path $\pi = (\ldots, g)$ with gate $g \in G$ and $g \to n$, we say that $\pi.n = (\ldots, g, n)$ is an *extension* of path π with node n.

Definition 6 (Path Selection)**.** Given a complete assignment $\sigma \in \mathcal{C}$ and a path $\pi = (\ldots, g)$ as above. *Gate input n can be selected* w.r.t. assignment σ to extend path π to $\pi.n$ if input n is controlling or if gate g has no controlling input.

Path selection based on the notion of controlling inputs as introduced above exploits observability don't cares as defined in the context of ATPG [82]. Similarly, we adopt the ATPG idea of backtracing [63, 82] as follows.

Definition 7 (Backtracing Step)**.** Given a complete assignment $\sigma \in \mathcal{C}$ and a gate $g \in G$ with $g \to n$. A *backtracing step* w.r.t. assignment σ selects gate input n

w.r.t. assignment σ as in Definition 6 and determines a *backtracing value* x for input n as follows. If $g = \neg n$, then $x = \sigma(g)$. Else, if $g = n \wedge m$, then $x = \neg\sigma(g)$.

As an important observation it turns out that performing a bit-level backtracing step always flips the value of the selected input under σ. For a selected input, the backtracing value is therefore always unique. This can be checked easily by considering all possible scenarios shown in Figure 3.

Proposition 8. *A backtracing step yields as backtracing value* $x = \neg\sigma(n)$.

Example 9. Consider $g = n \wedge m$ and the assignment $\sigma = \{g \mapsto 0, n \mapsto 0, m \mapsto 1\}$ consistent on g as depicted in Figure 3. Assume that $t = \neg\sigma(g) = 1$ is the target value of g, i.e., $\sigma(g) \rightsquigarrow t$ with $0 \rightsquigarrow 1$. We select n as the single controlling input of g (underlined), which yields backtracing value $x = \neg\sigma(n) = 1$.

A *trace* $\tau = (\pi, \alpha)$ is a rooted path $\pi = (n_1, \ldots, n_k)$ labelled with a partial assignment α, where α is defined exactly on $\{n_1, \ldots, n_k\}$. A trace (π, α) is (fully) *expanded*, if π is a fully expanded path, i.e., node $n_k \in V$ is the *leaf* of π and τ.

Let $\sigma \in \mathcal{C}\backslash\mathcal{W}$ be a complete consistent but non-satisfying assignment. Then the notion of extension is lifted from paths to traces as follows. Given a trace $\tau = (\pi, \alpha)$ with $\pi = (\ldots, g)$, $g \in G$ and $g \to n$. A backtracing (or *propagation*) step w.r.t. σ and target value $t = \neg\sigma(g) = \alpha(g)$ yields backtracing value $x = \neg\sigma(n) = \alpha'(n)$ and *extends* trace τ to $\tau' = (\pi', \alpha')$ (also denoted as $\tau \to \tau'$) if path $\pi' = \pi.n$ is an extension of π and $\alpha'(m) = \alpha(m)$ for all nodes m in π.

We define the *root trace* $\rho = ((r), \{r \mapsto 1\})$ as a trace that maps root r to its target value $\omega(r) = 1$. A *propagation trace* w.r.t. assignment σ is a (possibly partial) trace τ that starts from the root trace ρ and is extended by propagation steps w.r.t. assignment σ, denoted as $\rho \to^* \tau$.

Note that given a path π and σ, partial assignment α is redundant on the bit-level. However, we use the same notation on the word-level, where α captures updates to σ along π, which (in contrast to the bit-level) are not uniquely defined.

Definition 10 (Propagation Strategy)**.** Given a non-satisfying but consistent assignment $\sigma \in \mathcal{C}\backslash\mathcal{W}$, the set of valid moves $\mathcal{P}(\sigma)$ for σ under propagation strategy $\mathcal{P}$ contains exactly the leafs of all expanded propagation traces w.r.t σ.

In the following, we present and prove the main Lemma of this paper, which then immediately gives completeness of strategy $\mathcal{P}$ in Theorem 12. It is reused for proving completeness of the extension of $\mathcal{P}$ to the word-level in Theorem 25.

Lemma 11 (Propagation Lemma)**.** *Given a non-satisfying but consistent assignment* $\sigma \in \mathcal{C}\backslash\mathcal{W}$*, then for any satisfying assignment* $\omega \in \mathcal{W}$*, used as an oracle, there exists a fully expanded propagation trace* τ *w.r.t.* σ *with leaf* $v \in \Delta(\sigma, \omega)$.

Proof. The basic idea of our completeness proof is to inductively extend the root trace ρ to traces $\tau = (\pi, \alpha)$, i.e., $\rho \to^* \tau$, through propagation steps, which all satisfy the (key) invariant

$$\alpha(n) = \omega(n) \neq \sigma(n) \quad \text{for all nodes } n \text{ in } \pi. \tag{1}$$

The root trace $\rho = ((r), \{r \mapsto \omega(r)\})$ obviously satisfies this invariant. Now, let $\tau = (\pi, \alpha)$ be a trace that satisfies the invariant but is *not fully expanded*, i.e., $\pi = (r, \ldots, g)$ with $g \in G$ and $\alpha(g) = \omega(g) \neq \sigma(g)$. Since $\sigma(g) \neq \omega(g)$ it follows that g has at least one input n with $\sigma(n) \neq \omega(n)$. If g has no controlling input, then by Definition 6 it is allowed to select n as an input with $\sigma(n) \neq \omega(n)$. Otherwise, input n is selected as any controlling input. In both cases we select $x = \omega(n) \neq \sigma(n)$ as backtracing value using Proposition 8. Trace τ is now extended by n with backtracing value x to τ', i.e., $\tau \to \tau'$, which in turn concludes the inductive proof of Equation (1). Any fully expanded propagation trace $\tau = (\pi, \alpha)$ with leaf $v \in V$, as generated above, also satisfies the invariant in Equation (1). Thus, we have $\alpha(v) = \omega(v) \neq \sigma(v)$ with $v \in \Delta(\sigma, \omega)$. □

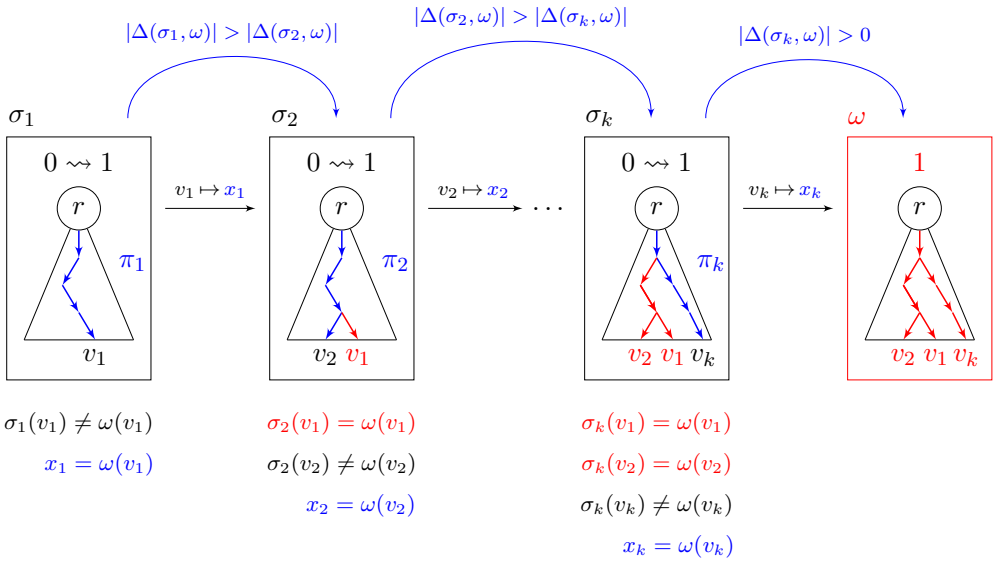


Figure 4: The basic idea of our completeness proof. Using a satisfying assignment ω as an oracle, in each move $\sigma_i \to \sigma_{i+1}$ we down-propagate target value $x = \omega(n)$ for all nodes n in propagation path $\pi_i = (r, \ldots, v_i)$, which yields update $x_i = \omega(v_i) = \sigma_{i+1}(v_i)$ and thus reduces the hamming distance $|\Delta(\sigma_i, \omega)| > |\Delta(\sigma_{i+1}, \omega)|$.

In essence, given assignment σ and ω as above, our propagation strategy propagates target value $\omega(r)$ from root r towards the primary inputs, ultimately producing a fully expanded propagation trace $\tau = (\pi, \alpha)$. In case of non-deterministic choices for extending the trace we use ω as an oracle to pick an input n with $\sigma(n) \neq \omega(n)$, which can be selected according to Definition 6. The oracle allows us to ensure that for all nodes $n \in \pi$, $\alpha(n) = \omega(n)$, which yields $\alpha(v) = \omega(v) \neq \sigma(v)$ and consequently $v \in \Delta(\sigma, \omega)$ for leaf v of τ. Thus, using Lemma 11, our propagation strategy turns out to be distance reducing, and therefore, according to Proposition 3, complete. Figure 4 illustrates the basic idea of our proof, which, in the following, serves as a basis for lifting our approach from the bit-level to the word-level.

Theorem 12. *Under the assumptions of the previous Lemma 11 we also get $v \in \mathcal{P}(\sigma)$ for leaf v. Thus, $\mathcal{P}$ is distance reducing and, as a consequence, complete.*

4 Word-Level

In the following, we only consider bit-vector expressions of fixed bit-width $w \in \mathbb{N}$. We denote a bit-vector expression n of width w as $n_{[w]}$, but will omit the bit-width if the context allows. We refer to the i-th bit of $n_{[w]}$ as $n[i]$ with $1 \leq i \leq w$ and, for the sake of simplicity, define bit indices as starting from 1 rather than 0. We interpret $n[1]$ as the least significant bit (LSB) and $n[w]$ as the most significant bit (MSB), and denote bit ranges over n from bit index j down to index i as $n[j:i]$. In string representations of bit-vectors, we interpret the bit at the far left index as the MSB and the bit at the far right index as the LSB. We further define *ctz* to be the common function that *counts the number of trailing zeroes* of a given bit-vector, i.e., the number of consecutive 0-bits starting from the LSB, e.g., $ctz(0101) = 0$ and $ctz(111100) = 2$. Similarly, *clz* is the common function to *compute the number of leading zeroes*, i.e., the number of consecutive 0-bits starting from the MSB, e.g., $clz(0101) = 1$ and $clz(111100) = 0$.

For the sake of simplicity and without loss of generality we consider a fixed single-rooted quantifier-free bit-vector formula ϕ and interpret Boolean expressions as bit-vector expressions of bit-width one. The set of bit-vector operators is restricted to $\mathcal{O} = \{\&, \sim, =, <, \ll, \gg, +, \cdot, \div, \text{mod}, \circ, [:], \text{if-then-else}\}$ and interpreted according to Table 1. The selection of operators in $\mathcal{O}$ is rather arbitrary but provides a good compromise between effective and efficient word-level rewriting and compact encodings for bit-blasting approaches. It is complete, though, in the sense that all operators defined in SMT-LIB [13] (in particular signed operators) can be modeled in a compact way. Note that our methods are not restricted to single-rootedness or this particular selection of operators, and can easily be lifted to any other set of operators or the multi-rooted case.

Operator	SMT-LIB	Arity	Bit-Width Output	Input			
$[j:i]$	`extract`	1	$j-i+1$	w	–	–	Extraction ($1 \leq i \leq j \leq w$)
$\sim$	`bvnot`	1	w	w	–	–	Bit-wise negation
$\&$	`bvand`	2	w	w	w	–	Bit-wise conjunction
$=$	`=`	2	1	w	w	–	Equality
$<$	`bvult`	2	1	w	w	–	Unsigned less than
$\ll$	`bvshl`	2	w	w	q	–	Logical shift left ($w = 2^q$)
$\gg$	`bvshr`	2	w	w	q	–	Logical shift right ($w = 2^q$)
$+$	`bvadd`	2	w	w	w	–	Addition
$\cdot$	`bvmul`	2	w	w	w	–	Multiplication
$\div$	`bvudiv`	2	w	w	w	–	Unsigned division
mod	`bvurem`	2	w	w	w	–	Unsigned remainder
$\circ$	`concat`	2	$p+q$	p	q	–	Concatenation
if-then-else	`ite`	3	w	1	w	w	Conditional

Table 1: The set of considered bit-vector operators ($w, p, q, i, j \in \mathbb{N}$).

We interpret formula ϕ as a single-rooted DAG represented as an 8-tuple $(r, N, \kappa, O, F, V, B, E)$. The set of *nodes* $N = O \cup V \cup B$ contains the single root node $r \in N$ of bit-width one, and is further partitioned into a set of *operator nodes* O, a set of *primary inputs* (or *bit-vector variables*) V, and a set of *bit-vector constants* $B \subseteq \mathbb{B}^*$, which are denoted in either decimal or binary notation if the context allows. The bit-width of a node is given by $\kappa\colon N \to \mathbb{N}$, thus $\kappa(r) = 1$. Operator nodes are interpreted as bit-vector operators via $F : O \to \mathcal{O}$, which in turn determines their arity and input and output bit-widths as defined in Table 1. The *edge* relation between nodes is given as $E = E_1 \cup E_2 \cup E_3$, with $E_i\colon O \to N^i$ describing the set of edges from unary, binary, and ternary operator nodes to its input(s), respectively. We again use the notation $o \to n$ for an edge between an operator node o and one of its inputs n.

We only consider well-formed formulas, where the bit-widths of all operator nodes and their inputs conform to the conditions imposed via interpretation F as defined in Table 1. For instance, we denote a bit-vector addition node o with inputs n and m as $o = n + m$, where $o \in O$ of arity 2 with $F(o) = +$, and therefore $\kappa(o) = \kappa(n) = \kappa(m)$. In the following, if more convenient we will use the functional notation $o = \diamond(n_1, \ldots, n_k)$ for operator node $o \in O$ of arity k with inputs $n_1, \ldots, n_k$ and $F(o) = \diamond$, e.g., $+(n, m)$. Note that the semantics of all operators in $\mathcal{O}$ correspond to their SMT-LIB counterparts listed in Table 1, with three exceptions. Given a logical shift operation $n \ll m$ or $n \gg m$, w.l.o.g. and as implemented in our SMT solver Boolector [91], we restrict bit-width $\kappa(n)$ to $2^{\kappa(m)}$. Further, as implemented by Boolector and other state-of-the-art SMT

solvers, e.g., Mathsat [42] Yices [53] and Z3 [49], we define an unsigned division by zero to return the greatest possible value rather than introducing uninterpreted functions, i.e., $x \div 0 = \sim 0$. Similarly, $x \bmod 0 = x$.

A *complete assignment* σ of a given fixed ϕ is a complete function $\sigma\colon N \to \mathbb{B}^*$ with $\sigma(n) \in \mathbb{B}^{\kappa(n)}$, and a *partial assignment* is a partial function $\alpha\colon N \to \mathbb{B}^*$ with $\alpha(n) \in \mathbb{B}^{\kappa(n)}$. Given an operator node $o \in O$ with $o = \diamond(n_1, \ldots, n_k)$ and $\diamond \in \mathcal{O}$, a complete assignment σ is *consistent* on o if $\sigma(o) = f(\sigma(n_1), \ldots, \sigma(n_k))$ where $f\colon \mathbb{B}^{\kappa(n_1)} \times \cdots \times \mathbb{B}^{\kappa(n_k)} \to \mathbb{B}^{\kappa(o)}$ is determined by the semantics of operator $\diamond$ as defined in the SMT-LIB standard [13] (with the exceptions discussed above).

A complete assignment is (globally) *consistent* on ϕ (or just *consistent*), iff it is consistent on all bit-vector operator nodes $o \in O$ and $\sigma(b) = b$ for all bit-vector constants $b \in B$. A *satisfying* assignment ω is a complete and consistent assignment that satisfies the root, i.e., $\omega(r) = 1$. In the following, we will again use the letter $\mathcal{C}$ to denote the set of complete and consistent assignments, and the letter $\mathcal{W}$ with $\mathcal{W} \subseteq \mathcal{C}$ to denote the set of satisfying assignments of formula ϕ.

Given a bit-vector variable $v \in V$ with $\kappa(v) = w$ and assignments $\sigma, \sigma' \in \mathcal{C}$. We adopt the notion of obtaining assignment σ' from assignment σ by assigning a new value x to variable v with $x \in \mathbb{B}^w$ and $x \neq \sigma(v)$, written as $\sigma \xrightarrow{v \mapsto x} \sigma'$, which we refer to as a *move*. The set of word-level moves is thus defined as $\mathcal{M} = \{(v, x) \mid v \in V, x \in \mathbb{B}^{\kappa(v)}\}$, and accordingly, a word-level propagation strategy $\mathcal{P}$ is defined as a function $S\colon \mathcal{C} \mapsto \mathbb{P}(\mathcal{M})$, which maps a consistent assignment to a set of moves. We lift propagation strategy $\mathcal{P}$ from the bit-level to the word-level by first introducing our new notion of *essential inputs*, which lifts and extends the bit-level notion of controlling inputs to the word-level.

Definition 13 (Essential Inputs)**.** Let $n \in N$ be an input of a bit-vector operator node $o \in O$, i.e., $o \to n$, and let σ be a complete assignment consistent on o. Further, let t be the *target value* of o, i.e., $\sigma(o) \rightsquigarrow t$, with $t \neq \sigma(o)$. We say that n is an *essential input* under σ w.r.t. target value t, if for all complete assignments σ' consistent on o with $\sigma(n) = \sigma'(n)$, we have $\sigma'(o) \neq t$.

In other words, an input n to an operator node o is *essential* w.r.t. some target value t, if o can not assume t as long as the assignment of n does not change. As an example, consider the bit-vector operators and their essential inputs under some consistent assignment σ w.r.t. some target value t as depicted in Figure 5.

Example 14. Consider the bit-vector operators $\{+, \,\&, \ll, \cdot, \div, \bmod, \circ\}$ of bit-width 2 as depicted in Figure 5. For an operator node o, at the outputs we denote given assignment $\sigma(o)$ and target value t as $\sigma(o) \rightsquigarrow t$ (e.g., $10 \rightsquigarrow 01$). At the inputs we show their assignment under σ. Essential inputs (under σ w.r.t. target value t) are indicated with an underline.

(a) Given $\mathbf{o} := \mathbf{n} + \mathbf{m}$ with $t = 10$ and $\sigma = \{o \mapsto 11,\ n \mapsto 00,\ m \mapsto 11\}$. Operator + has no essential inputs, independent from σ and t.

(b) Given $\mathbf{o} := \mathbf{n}$ & $\mathbf{m}$ with $t = 01$ and $\sigma = \{o \mapsto 10,\ n \mapsto 10,\ m \mapsto 11\}$.
Input n is essential since t & $\sigma(n) \neq t$ and thus, it is not possible to find a value x for m such that $\sigma(n)$ & $x = t$. Input m, however, is not essential since t & $\sigma(m) = t$.

(c) Given $\mathbf{o} := \mathbf{n} \ll \mathbf{m}$ with $t = 10$ and $\sigma = \{o \mapsto 00,\ n \mapsto 00,\ m \mapsto 1\}$.
Input n is obviously essential, since shifting zero value 00 can never result in the non-zero target value $t = 01$. Input m, however, is not essential, since it is possible to simply select, e.g., $x = 01$ for n such that $t = 10 = x \ll \sigma(m) = 01 \ll 1$.

(d) Given $\mathbf{o} := \mathbf{n} \cdot \mathbf{m}$ with $t = 10$ and $\sigma = \{o \mapsto 00,\ n \mapsto 00,\ m \mapsto 10\}$.
Input n is essential since $t \neq 00$ but $\sigma(n) = 00$, and thus, it is not possible to find a value x for m such that $\sigma(n) \cdot x = t$. Input m, however, is not essential since we could pick, e.g., $x = 01$ for n to obtain $t = 10 = x \cdot \sigma(m) = 01 \cdot 10$.

(e) Given $\mathbf{o} := \mathbf{n} \div \mathbf{m}$ with $t = 10$ and $\sigma = \{o \mapsto 01,\ n \mapsto 01,\ m \mapsto 01\}$.
Input n is essential since $\sigma(n) < t$, and thus, it is not possible to find a value x for m such that $\sigma(n) \div x = t$. Input m, however, is not essential, since we could pick, e.g., $x = 10$ to obtain $t = 10 = x \cdot \sigma(m) = 10 \cdot 01$.

(f) Given $\mathbf{o} := \mathbf{n}$ mod $\mathbf{m}$ with $t = 10$ and $\sigma = \{o \mapsto 00,\ n \mapsto 01,\ m \mapsto 01\}$.
Since $\sigma(n) = 01 < 10 = t$, it is not possible to find a value x for m such that $\sigma(n) \bmod x = t$. However, since $\sigma(m) = 01$ but $t \neq 00$, it is also not possible to find a value x for n such that $x \bmod \sigma(m) = t$. Hence, both inputs are essential.

(g) Given $\mathbf{o} := \mathbf{n} \circ \mathbf{m}$ with $t = 11$ and $\sigma = \{o \mapsto 11,\ n \mapsto 0,\ m \mapsto 1\}$.
Input n is essential since $\sigma(n) \neq t[2:2]$, and thus, it is not possible to find a value x for m such that $\sigma(n) \circ x = t$. Input m, however, is not essential since it already matches the corresponding slice of the target value.

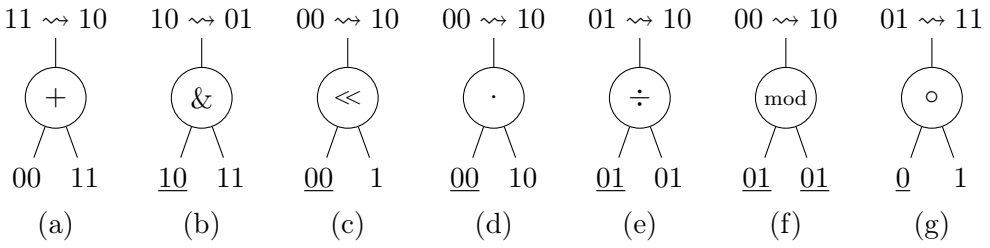


Figure 5: Bit-vector operator nodes and examples for essential (underlined) inputs. Given output values indicate the transition from current to target value.

Note that AIGs can be represented by bit-vectors of bit-width one, which can be interpreted as Boolean expressions. In this sense, the notion of controlling inputs can also be applied to word-level Boolean expressions.

Proposition 15. *When applied to bit-level (Boolean) expressions, the notion of* essential inputs *exactly matches the notion of* controlling inputs.

Proof. For applying the notion of essential inputs to the bit-level, consider the operator set $\{\neg, \wedge\}$ for $o \in G$ with $o \to n$. Further, target value $t \neq \sigma(o)$ on the bit-level implies $t = \neg\sigma(o)$, which exactly matches the corresponding implicit definition of the target value of o on the bit-level. Now assume that input n is essential w.r.t. t. Then, if $\sigma(n) = \sigma'(n)$, by Definition 13 we have that $\sigma'(o) \neq t$, and therefore $\sigma'(o) = \neg t = \sigma(o)$. The other direction works in the same way. □

The definition of a (rooted and expanded) *path* as a sequence of nodes $\pi = (n_1, \ldots, n_k) \in N^*$ is lifted from the bit-level to the word-level in the natural way. Corresponding restrictions and implications of Section 3 apply. The notions of *path selection* and *path extension* are lifted to the word-level as follows.

Definition 16 (Path Extension)**.** Given a path $\pi = (\ldots, o)$ with $o \in O$ and $o \to n$, we say that $\pi.n = (\ldots, o, n)$ is an extension of path π with node n.

Definition 17 (Path Selection)**.** Given a complete consistent assignment $\sigma \in \mathcal{C}$, a path $\pi = (\ldots, o)$ as in Definition 16 above, and $\sigma(o) \rightsquigarrow t$, i.e., $t \neq \sigma(o)$, then *input n can be selected* w.r.t. σ and target value t to extend π to $\pi.n$ if n is essential or if o has no essential input (in both cases essential under σ w.r.t. t).

Figure 6 shows examples for all combinations of essential (underlined) and non-essential inputs for all bit-vector operators in $\mathcal{O}$. For an operator node o, an output value $\sigma(o) \rightsquigarrow t$ indicates the desired transition from current assignment $\sigma(o)$ to target value t, and an input value shows its assignment under σ. Underlined blue cases indicate that this input is a single essential input and will therefore always be selected. Any other case (both inputs are essential or no input is essential) represents a non-deterministic choice during path selection.

In contrast to value selection on the bit-level, where a backtracing step always yields the flipped assignment of the selected input as backtracing value, on the word-level, selecting a backtracing value is not uniquely defined but a source of non-determinism. We consider three variants of value selection, under the following assumptions. Let t be the target value of an operator node $o \in O$, and let $\sigma \in \mathcal{C}$ be a complete assignment such that $\sigma(o) \neq t$. Further, assume that input n with $o \to n$ is selected w.r.t. target value t and σ as in Definition 17 above.

Definition 18 (Random Value)**.** Any value x with $\kappa(x) = \kappa(n)$ is called a *random value* for input n.

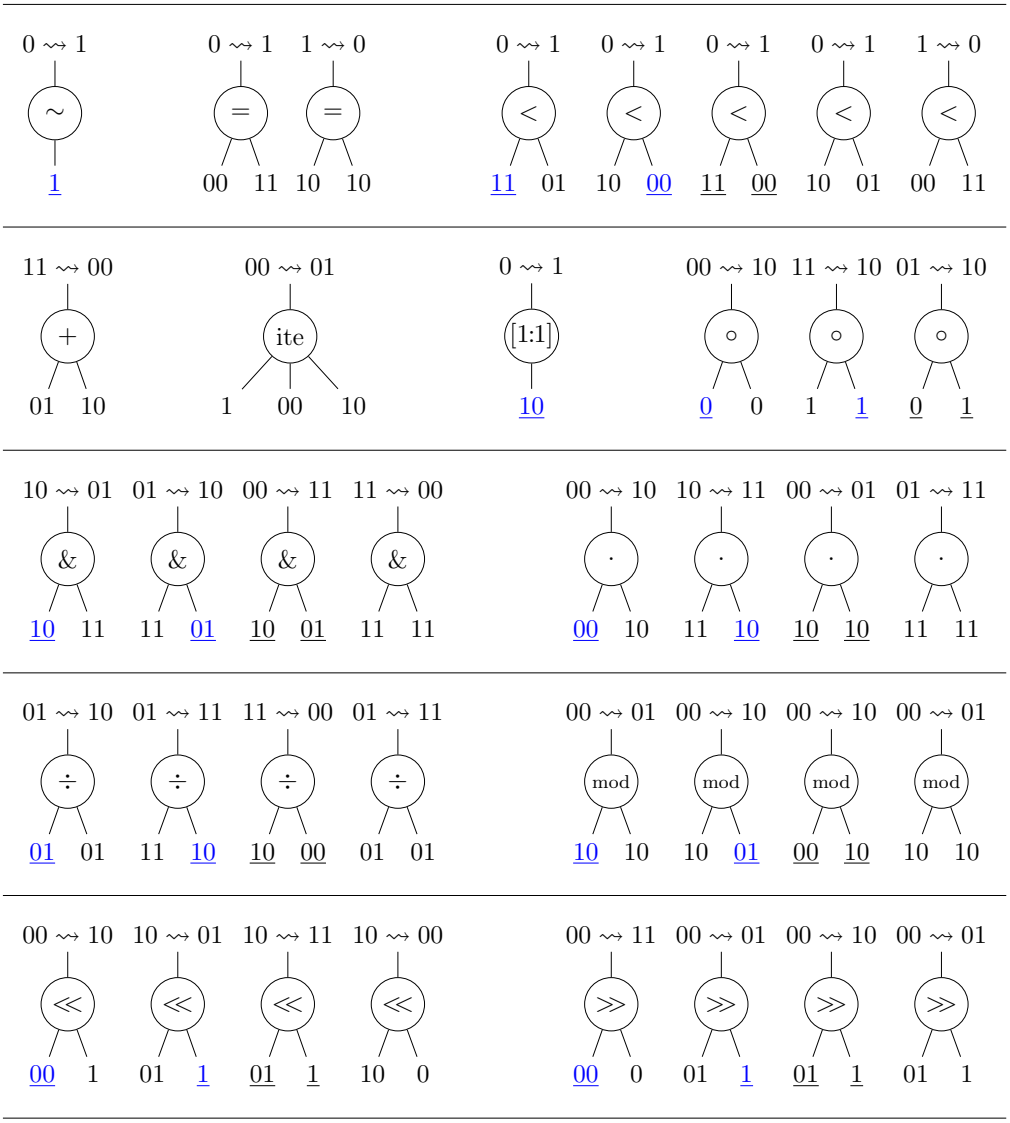


Figure 6: Examples for all combinations of essential (underlined) inputs for all bit-vector operators in $\mathcal{O}$. Underlined blue cases indicate that this input is a single essential input. No input is essential for operators $=$, $+$, and if-then-else.

Definition 19 (Consistent Value)**.** A random value x is a *consistent value* for input n w.r.t. target value t, if there is a complete assignment σ' consistent on operator node o with $\sigma'(n) = x$ and $\sigma'(o) = t$.

In other words, a value is consistent for an input, if it allows to produce the target value after changing values of other inputs if necessary. We compute a consistent value as backtracing value x for input n as described in Tables 2-3. However, in some cases, restricting the notion of consistent values even further may be beneficial. Consider the following motivating example.

Example 20. Consider a formula $\phi := 274177_{[65]} \cdot v = 18446744073709551617_{[65]}$. Computing $x = 18446744073709551617_{[65]} \div 274177_{[65]} = 67280421310721_{[65]}$ immediately concludes with a satisfying assignment for ϕ.

The chances to select $x = 67280421310721_{[65]}$ if consistent values for the multiplication operator are chosen as described in Table 2 are arbitrarily small. Hence, we also consider the notion of *inverse values*, which utilize the inverse of a given operator.

Definition 21 (Inverse Value)**.** A consistent value x is an *inverse value* for input n w.r.t. target value t and assignment σ, if there exists a complete assignment σ' consistent on operator node o with $\sigma'(n) = x$, $\sigma'(o) = t$ and $\sigma'(m) = \sigma(m)$ for all inputs m with $o \to m$ and $m \neq n$.

In other words, a value is an inverse value for input n, if it allows to produce the target value for an operator node without changing the assignment of its other inputs. Consequently, an inverse value for input n is also consistent. We compute an inverse value as backtracing value x for input n as described in Tables 4-6.

Note that inverse value computation as initially presented in [93] is too restrictive for some operators, which is incomplete since it may inadvertently prune the search. We therefore require that inverse value computation allows to generate all possible values for all operators in $\mathcal{O}$, which is the case for the rules for inverse value computation as described in Tables 4-6.

Definition 22 (Backtracing Step)**.** Let $\sigma \in \mathcal{C}$ be a complete consistent assignment. Given an operator node $o \in O$ with $o \to n$ and a target value $t \neq \sigma(o)$, then a *backtracing step* selects input n of operator node o w.r.t. σ as in Definition 17 and selects a *backtracing value* x for n as a *consistent* (and optionally *inverse*) value w.r.t. σ and t if such a value exists, and a *random value* otherwise.

Note that it is not always possible to find an inverse value for input n, e.g., $o := n \,\&\, m$ with $\sigma = \{o \mapsto 00, n \mapsto 00, m \mapsto 00\}$ and $t = 01$. Further, even for operators that allow to always produce inverse values, e.g., operator $+$, doing so may lead to inadvertently pruning the search space, see Example 23 below.

$\mathbf{o} \coloneqq \sim \mathbf{n}$	Then $x = \sim t$.
$\mathbf{o} \coloneqq \mathbf{n} = \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} = \mathbf{n}$	Any random value x is consistent for operator $=$.
$\mathbf{o} \coloneqq \mathbf{n} \ \& \ \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} \ \& \ \mathbf{n}$	Let i be a bit index with $1 \leq i \leq \kappa(n)$. For all i, if $t[i] = 1$ then $x[i] = 1$, and else, $x[i]$ is set arbitrarily.
$\mathbf{o} \coloneqq \mathbf{n} < \mathbf{m}$	Any random value x is a consistent value for n, with the restriction that if $t = 1$ then $x < \sim 0$.
$\mathbf{o} \coloneqq \mathbf{m} < \mathbf{n}$	Any random value x is a consistent value for n, with the restriction that if $t = 1$ then $x \neq 0$.
$\mathbf{o} \coloneqq \mathbf{n} + \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} + \mathbf{n}$	Any x is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \cdot \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} \cdot \mathbf{n}$	Any random value x with $ctz(t) \geq ctz(x)$ and $x = 0$ if $t = 0$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \div \mathbf{m}$	If $t = \sim 0$ or $t = 0$, any arbitrary value x is consistent, with the restriction that $x < \sim 0$ if $t = 0$. In any other case, let y be a random value with $y \neq 0$ such that $y \cdot t$ does not overflow. Then $x = y \cdot t$.
$\mathbf{o} \coloneqq \mathbf{m} \div \mathbf{n}$	If $t = \sim 0$, then $x \in \{0, 1\}$ is a consistent value for n. Else, any random value x such that $x \cdot t$ does not overflow is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \bmod \mathbf{m}$	If $t = \sim 0$ then $x = \sim 0$, and a random $x \geq t$, otherwise.
$\mathbf{o} \coloneqq \mathbf{m} \bmod \mathbf{n}$	If $t = \sim 0$ then $x = 0$, and a random $x > t$, otherwise.
$\mathbf{o} \coloneqq \mathbf{n} \ll \mathbf{m}$	Let s be a random value with $0 \leq s \leq ctz(t)$, and let $w = \kappa(n)$. Then $x[i] = (t \gg s)[i]$ for $1 \leq i \leq w - s$, and all other bits $x[i]$ set arbitrarily for $w - s < i \leq w$.
$\mathbf{o} \coloneqq \mathbf{m} \ll \mathbf{n}$	Any x with $0 \leq x \leq ctz(t)$ is a consistent value for n.

Table 2: Rules for consistent value computation for operators $\{\sim, =, \&, <, +, \cdot, \div, \text{mod}, \ll\}$, where t is the target value of an operator node $o \in O$, $\sigma \in \mathcal{C}$ is a complete assignment such that $\sigma(o) \neq t$, and input n with $o \to n$ is selected w.r.t. target value t and assignment σ as in Definition 17.

$\mathbf{o} \coloneqq \mathbf{n} \gg \mathbf{m}$	Let s be a random value with $0 \leq s \leq clz(t)$, and let $w = \kappa(n)$. Then $x[i] = (t \ll s)[i]$ for $s < i \leq w$, and all other bits $x[i]$ set arbitrarily for $1 \leq i \leq s$.
$\mathbf{o} \coloneqq \mathbf{m} \gg \mathbf{n}$	Any x with $0 \leq x \leq clz(t)$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n} \circ \mathbf{m}$	Let $p = \kappa(n)$ and $q = \kappa(m)$ and $w = \kappa(o) = p + q$. Then $x = t[w : q + 1]$ is a consistent value for n.
$\mathbf{o} \coloneqq \mathbf{m} \circ \mathbf{n}$	Let $p = \kappa(n)$ and $q = \kappa(m)$ and $w = \kappa(o) = p + q$. Then $x = t[q : 1]$ is consistent value for n.
$\mathbf{o} \coloneqq \mathbf{n}[\mathbf{j} : \mathbf{i}]$	Then $x[k] = t[k + i - 1]$ for $1 \leq k \leq j - i + 1$, with all other bits set arbitrarily.
$\mathbf{o} \coloneqq$ if $\mathbf{c}$ then $\mathbf{n}$ else $\mathbf{m}$ $\mathbf{o} \coloneqq$ if $\mathbf{c}$ then $\mathbf{m}$ else $\mathbf{n}$ $\mathbf{o} \coloneqq$ if $\mathbf{n}$ then $\mathbf{m_1}$ else $\mathbf{m_2}$	Any x is a consistent value for n.

Table 3: Rules for *consistent* value computation for operators $\{\gg, \circ, [:],$ if-then-else$\}$, where t is the target value of an operator node $o \in O$, $\sigma \in \mathcal{C}$ is a complete assignment such that $\sigma(o) \neq t$, and input n with $o \to n$ is selected w.r.t. target value t and assignment σ as in Definition 17.

$\mathbf{o} \coloneqq \sim \mathbf{n}$	Then $x = \sim t$.
$\mathbf{o} \coloneqq \mathbf{n} = \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} = \mathbf{n}$	If $t = 1$, then $x = \sigma(m)$. In any other case, any random value $x \neq \sigma(m)$ is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{n}$ & $\mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m}$ & $\mathbf{n}$	Let i be a bit index with $1 \leq i \leq \kappa(n)$. If there is an i with $t[i] = 1$ and $\sigma(m)[i] = 0$, then there exists no inverse value for n. Otherwise, for all i, if $t[i] = 1$ then $x[i] = 1$, or if $t[i] = 0$ and $\sigma(m)[i] = 1$ then $x[i] = 0$, and else, $x[i]$ is set arbitrarily.
$\mathbf{o} \coloneqq \mathbf{n} < \mathbf{m}$	If $t = 1$ and $\sigma(m) = 0$, then there exists no inverse value. Else, any x with $t = x < \sigma(m)$ is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{m} < \mathbf{n}$	If $t = 1$ and $\sigma(m) = \sim 0$, then there exists no inverse value. Else, any x with $t = \sigma(m) < x$ is an inverse value for n.

Table 4: Rules for *inverse* value computation for operators $\{\sim, =, \&, <\}$, where t is the target value of an operator node $o \in O$, $\sigma \in \mathcal{C}$ is a complete assignment such that $\sigma(o) \neq t$, and input n with $o \to n$ is selected w.r.t. target value t and assignment σ as in Definition 17.

$\mathbf{o} \coloneqq \mathbf{n} + \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} + \mathbf{n}$	Then $x = t - \sigma(m) = t + (1 + {\sim}\sigma(m))$.
$\mathbf{o} \coloneqq \mathbf{n} \cdot \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} \cdot \mathbf{n}$	If $t = \sigma(m) = 0$, any x is an inverse value, but this contradicts assumption $t \neq \sigma(o)$. If $t \neq 0$ and $\sigma(m) = 0$, or if $\sigma(m) \neq 0$ with $ctz(t) < ctz(\sigma(m))$, there exists no inverse value. Otherwise, $ctz(t) \geq ctz(\sigma(m))$ and $\sigma(m) \neq 0$. Let $y = m \gg ctz(\sigma(m))$, thus y is odd. We compute y^{-1} as its multiplicative inverse modulo 2^w, e.g., via the Extended Euclidean algorithm (similar to word-level rewriting techniques that require solving for a variable, e.g. [60]), and determine x as $(t \gg ctz(\sigma(m))) \cdot y^{-1}$ except that all bits in $x[w : w - ctz(\sigma(m)) + 1]$ are set arbitrarily, with $w = \kappa(n)$.
$\mathbf{o} \coloneqq \mathbf{n} \div \mathbf{m}$	If $t = {\sim}0$ and $\sigma(m) = 0$, then any x is an inverse value, but this contradicts assumption $t \neq \sigma(o)$. If $t = {\sim}0$ and $\sigma(m) \notin \{0, 1\}$, or if $t \neq {\sim}0$ and $\sigma(m) = 0$, or if $t \cdot \sigma(m)$ produces an overflow, there exists no inverse value for n. Else, if $t = {\sim}0$ and $\sigma(m) = 1$, then $x = {\sim}0$. In any other case, any x with $t = x \div \sigma(m)$ is an inverse value.
$\mathbf{o} \coloneqq \mathbf{m} \div \mathbf{n}$	If $t = \sigma(m) = 0$, then any x is an inverse value, but this contradicts assumption $t \neq \sigma(o)$. If $t = 0$ and $\sigma(m) = {\sim}0$, or if $m < t$, then there exists no inverse value for n. If $t = \sigma(m) = {\sim}0$, then $x \in \{0, 1\}$, and if $t = {\sim}0$ and $\sigma(m) \neq {\sim}0$, then $x = 0$. Else, if $t = 0$ and $\sigma(m) \neq {\sim}0$, then any random $x > \sigma(m)$ is an inverse value. In any other case, any x with $t = \sigma(m) \div x$ is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{n} \bmod \mathbf{m}$	If $\sigma(m) \leq t$, then there exists no inverse value. Else, we select a $y \neq 0$ such that neither in the multiplication nor the addition operation of $\sigma(m) \cdot y + t$ occurs an overflow. Then $x = \sigma(m) \cdot y + t$ is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{m} \bmod \mathbf{n}$	If $\sigma(m) < t$, or if $t \neq 0$ and $t = \sigma(m) - 1$, or if $\sigma(m) - t \leq t$, then there exists no inverse value for n. Else, if $\sigma(m) = t$, then $x = 0$ or any $x > t$ is an inverse value for n. In any other case, any $x = (\sigma(m) - t) \div y$ with $y > 0$ such that $(\sigma(m) - t) \bmod y = 0$ is an inverse value for n.

Table 5: Rules for *inverse* value computation for operators $\{+, \cdot, \div, bmod\}$, where t is the target value of an operator node $o \in O$, $\sigma \in \mathcal{C}$ is a complete assignment such that $\sigma(o) \neq t$, and input n with $o \to n$ is selected w.r.t. target value t and assignment σ as in Definition 17.

$\mathbf{o} \coloneqq \mathbf{n} \ll \mathbf{m}$	If $\sigma(m) = 0$, then any x is an inverse value. Else, if $ctz(t) \geq ctz(m)$, then $x = t \gg \sigma(m)$ with all bits in $x[w : w-\sigma(m)+1]$ with $w = \kappa(n)$ set arbitrarily. In any other case, there exists no inverse value for n.
$\mathbf{o} \coloneqq \mathbf{m} \ll \mathbf{n}$	If $t = \sigma(m) = 0$, then any x is an inverse value, but this contradicts assumption $t \neq \sigma(o)$. If $ctz(m) \leq ctz(t)$, if $t = 0$, any $x \geq ctz(t) - ctz(m)$ is an inverse value for n, and else, $x = ctz(t) - ctz(m)$ is an inverse value if the remaining shifted bits in t match with the corresponding bits in $\sigma(m)$, i.e., if $t[w : x+1] = \sigma(m)[w-x : 1]$ with $w = \kappa(n) = \kappa(o)$. In any other case, there exists no inverse value for n.
$\mathbf{o} \coloneqq \mathbf{n} \gg \mathbf{m}$	If $\sigma(m) = 0$, then any x is an inverse value. Else, if $clz(t) \geq clz(m)$ then $x = t \ll \sigma(m)$ with all bits in $x[\sigma(m) : 1]$ set arbitrarily. In any other case, there exists no inverse value for n.
$\mathbf{o} \coloneqq \mathbf{m} \gg \mathbf{n}$	If $t = \sigma(m) = 0$, then any x is an inverse value, but this contradicts assumption $t \neq \sigma(o)$. If $clz(m) \geq clz(t)$, if $t = 0$, then any $x \geq clz(t) - clz(m)$ is an inverse value for n, and else, $x = clz(t) - clz(m)$ is an inverse value, if the remaining shifted bits in t match with the corresponding bits in $\sigma(m)$, i.e., if $t[w-x : 1] = \sigma(m)[w : x+1]$ with $w = \kappa(m) = \kappa(o)$. In any other case, there exists no inverse value for n.
$\mathbf{o} \coloneqq \mathbf{n} \circ \mathbf{m}$ $\mathbf{o} \coloneqq \mathbf{m} \circ \mathbf{n}$	Then any consistent value x is an inverse value for n.
$\mathbf{o} \coloneqq \mathbf{n}[\mathbf{j} : \mathbf{i}]$	Then any consistent value x is an inverse value for n.
$\mathbf{o} \coloneqq$ if $\mathbf{c}$ then $\mathbf{n}$ else $\mathbf{m}$ $\mathbf{o} \coloneqq$ if $\mathbf{c}$ then $\mathbf{m}$ else $\mathbf{n}$	Then $x = t$.
$\mathbf{o} \coloneqq$ if $\mathbf{n}$ then $\mathbf{m_1}$ else $\mathbf{m_2}$	Then $x = \sim\sigma(n)$

Table 6: Rules for *inverse* value computation for operators $\{\ll, \gg, \circ, [:],$ if-then-else$\}$, where t is the target value of an operator node $o \in O$, $\sigma \in \mathcal{C}$ is a complete assignment such that $\sigma(o) \neq t$, and input n with $o \to n$ is selected w.r.t. target value t and assignment σ as in Definition 17.

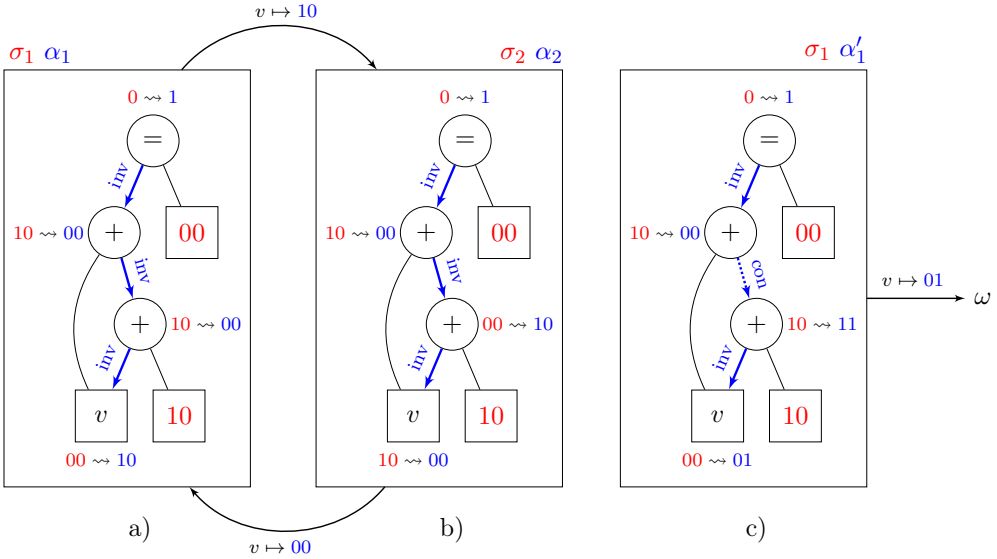


Figure 7: Example illustrating the necessity of choosing between random and inverse values when down propagating assignments (backtracing). The output values indicate the (desired) transition from current to target value. Other values indicate the transition from current value to the value yielded by down propagating the target output value. A propagation strategy without further randomization using only inverse values is incomplete.

Example 23. Consider formula $\phi := v + v + 2_{[2]} = 0_{[2]}$ with root $r := p_2 = 0_{[2]}$, where $p_2 := v + p_1$ and $p_1 := v + 2_{[2]}$, and a complete consistent assignment $\sigma_1 = \{v \mapsto 00,\ p_1 \mapsto 10,\ p_2 \mapsto 10,\ r \mapsto 0\}$, as shown in Figure 7a. Let $t = 1$ be the target value of root r, i.e., our goal is to find a value for bit-vector variable v such that $p_2 = 00$, and thus, formula ϕ is satisfied. Assume that as in Figure 7a-b, only inverse values are selected for $+$ operators during propagation. Down propagating target values along the path indicated by blue arrows in Figure 7a, the move $v \mapsto 10 = \alpha_1(v)$ is generated, which consequently yields assignment $\sigma_2 = \{v \mapsto 10,\ p_1 \mapsto 00,\ p_2 \mapsto 10,\ r \mapsto 0\}$ as indicated in Figure 7b. Selecting the other possible propagation path, the same move is produced. Thus, σ_2 is independent of which of the two paths is selected. Since $\sigma_2(r) \neq t$, target value t is again propagated down, which generates move $v \mapsto 00 = \alpha_2(v)$, again independently of which path is selected. With this, we move back to the initial assignment σ_1. Consequently, a satisfying assignment, e.g., $\omega(v) = 01$ or $\omega'(v) = 11$, can not be reached by only selecting inverse values. However, selecting a consistent but non-inverse value for p_1 as, e.g., in Figure 7c, generates move $v \mapsto 01 = \alpha'_1(v)$, which yields a satisfying assignment $\omega = \{v \mapsto 01,\ p_1 \mapsto 11,\ p_2 \mapsto 00,\ r \mapsto 1\}$.

As a consequence, when performing a backtracing step we in general select some consistent non-inverse value, if no inverse value exists, and otherwise non-deterministically choose between consistent (but not necessarily inverse) and inverse values. Since all operators in $\mathcal{O}$ are *surjective* (i.e., they can produce any target value) for our selected semantics (e.g., $\sim 0 \bmod 0 = \sim 0$), it is not necessary to select inconsistent random values. For other sets of operators, however, this might be necessary. For the sake of completeness we therefore included the selection of random values in the formal definition of backtracing steps.

Note that since on the bit-level the backtracing value for a selected input is uniquely determined (see Proposition 8), the issue of value selection is specific to the word-level. Further, when interpreting AIGs as word-level expressions, the notion of backtracing steps on the bit-level as in Definition 7 exactly matches the word-level notion as in Definition 22 using Proposition 15. As a side note, the problem of value selection during word-level backtracing and subsequent word-level propagation is similar to the problem of making a theory decision ("model assignment") and propagating this decision in MCSat [50, 76].

The *word-level propagation strategy* $\mathcal{P}$ is defined in exactly the same way as for the bit-level (see Definition 10) except that the word-level notion of backtracing based on essential inputs and consistent and inverse value selection (Definition 22) replaces bit-level backtracing based on controlling inputs (Definition 7), and the set of valid moves $\mathcal{P}(\sigma)$ contains not only the leafs of all expanded propagation traces but also their updated assignments, i.e., $(v, \alpha(v))$ for a leaf v. Further important concepts defined on the bit-level in Section 3 can be extended naturally to the word-level. These concepts include (expanded) paths and traces, leafs, and trace extension. We omit formal definitions accordingly.

Proposition 8, which is substantial for the bit-level proof of Lemma 11, does not directly apply on the word-level due to the more sophisticated selection of backtracing values. We lift Proposition 8 to the word-level as follows.

Proposition 24. *Let $\sigma \in \mathcal{C}$ be a complete consistent assignment, and let ω be a satisfying assignment $\omega \in \mathcal{W}$. Given operator node $o \in O$ with $o \to n$ and target value $t = \omega(o) \neq \sigma(o)$, i.e., $\sigma(o) \rightsquigarrow t$, then there exists a backtracing step w.r.t. assignment σ and target value t, which selects input n and backtracing value $x = \omega(n) \neq \sigma(n)$.*

Proof. First, assume that operator node o has an essential input w.r.t. assignment σ. Then we select an arbitrary essential input n of o. Since target value $t = \omega(o) \neq \sigma(o)$, we get $\sigma(n) \neq \omega(n)$ by contraposition of Definition 13. Similarly, if o has no essential inputs, then we select n as an arbitrary input with $\sigma(n) \neq \omega(n)$, which has to exist since $\omega(o) \neq \sigma(o)$. In both cases, we can select $x = \omega(n) \neq \sigma(n)$ as backtracing value, which is consistent for operator node o w.r.t. assignment σ and target value t since ω is consistent. Picking a random value as backtracing value, which is the last case in Definition 22, can not occur under the given assumptions since, as already discussed, ω is consistent on o. □

Using Proposition 24 instead of Proposition 8, the bit-level proof of Lemma 11 can then be lifted to the word-level by replacing every occurrence of gate g with operator node o, and the notion of "controlling" input with "essential" input.

Theorem 25. *Theorem 12 and Lemma 11 also apply on the word-level, and thus, propagation strategy $\mathcal{P}$ is also complete on the word-level.*

Note that even though Proposition 24 would allow us to restrict the selection of consistent and inverse backtracing values to be different from the current input node value, i.e., $x \neq \sigma(n)$, we do not enforce this property. Restricting value selection to a value $x \neq \sigma(n)$ interferes with path selection, in particular in the case where an input node is selected for which the current value is the only consistent or inverse value. We leave the exploration of this optimization to future work.

5 Experimental Evaluation

We implemented our propagation strategy within our SMT solver Boolector [91] and consider the following configurations.

(1) **Bb** The core Boolector engine which implements a bit-blasting approach. This configuration is identical to the version that entered the QF_BV track of the SMT competition 2016 and uses (internal) version bbc of our SAT solver Lingeling [23] as backend solver.

(2) **Bsls** The SLS engine of Boolector, which implements the SLS for SMT approach introduced in [58] as described in [93], with random walks enabled.

(3) **Paig** The bit-level configuration of our propagation-based approach which operates on AIG representation of the given input as bit-blasted by Boolector.

(4) **Pw** The word-level configuration of our propagation-based approach which directly operates on the given bit-vector formula, with inverse values prioritized over consistent values during backtracing with a probability of 99 to 1.

Note that the choice of rewriting and other simplification techniques applied prior to the actual decision procedure may considerably influence its performance. In order to provide the same basis for comparison and avoid skewed results due to differences in the rewriting and simplification techniques applied by Z3 [49] versus Boolector, we do not compare our propagation-based approach against the original implementation of [58] in Z3 but against our implementation of [58] in Boolector (configuration Bsls). All configurations of Boolector apply the same set of rewriting and simplification techniques in the same order.

	Bb		Bsls		Paig		Pw	
Family	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
asp (376)	**48**	**3526.2**	0	3760.0	0	3760.0	0	3760.0
bench_ab (223)	223	0.2	**223**	**0.0**	**223**	**0.0**	**223**	**0.0**
bmc (22)	**20**	**58.5**	10	122.2	11	134.3	13	116.3
brummayerbiere (26)	5	228.2	25	13.5	12	184.8	**26**	**17.5**
calypto (13)	4	92.4	4	91.1	2	110.2	**5**	**93.3**
check2 (1)	1	0.0	1	0.0	1	0.0	1	0.0
crafted (1)	1	0.0	1	0.0	1	0.0	1	0.0
dwp_formulas (103)	103	0.4	**103**	**0.0**	**103**	**0.0**	**103**	**0.0**
fft (19)	**4**	**159.2**	0	190.0	0	190.0	1	180.5
float (126)	21	1124.9	0	1260.0	0	1260.0	**27**	**1033.7**
gulwani (6)	**5**	**28.3**	1	51.0	0	60.0	0	60.0
mcm (155)	**14**	**1477.1**	5	1523.1	5	1528.0	12	1440.2
pspace (21)	0	210.0	21	17.3	0	210.0	**21**	**1.6**
rubik (3)	**1**	**23.1**	0	30.0	0	30.0	0	30.0
RWS (20)	**14**	**84.5**	0	200.0	0	200.0	0	200.0
sage (6236)	**6236**	**2602.9**	5287	11117.4	4623	17036.3	5099	11615.9
Sage2 (6981)	**1564**	**60634.3**	613	64540.2	289	67648.1	526	64762.5
spear (1675)	1395	10587.1	1145	6848.8	1516	3516.0	**1668**	**205.2**
stp (1)	0	10.0	0	10.0	0	10.0	0	10.0
stp_samples (149)	**149**	**4.4**	120	523.4	129	217.9	104	546.5
tacas07 (3)	**3**	**11.5**	2	10.2	2	11.3	2	10.2
uclid (262)	261	741.1	2	2610.1	28	2467.0	**262**	**148.6**
VS3 (10)	0	100.0	0	100.0	0	100.0	0	100.0
wienand (4)	0	40.0	0	40.0	0	40.0	0	40.0
total (16436)	**10072**	**81744.5**	7560	93058.3	6945	98714.1	8094	84372.0

Table 7: Configurations Bb, Bsls, Paig and Pw with a time limit of 10 seconds grouped by benchmark families. Configuration Bb is the default bit-blasting engine of Boolector. Configuration Bsls implements the SLS for SMT approach presented in [58] in Boolector as described in [93]. Configurations Paig and Pw implement our bit-level and word-level propagation strategy.

Since [92] and in particular for the SMT competition 2016, we improved several core components of Boolector, which affects all the configurations above. The default configurations of Paig and Pw therefore show major improvements in comparison to [92]. In comparison to [93], the default configuration of Bsls, however, seems to perform worse. This is solely due to minor changes within the SLS engine that affect the random number generator (RNG). We will show that the difference in the number of solved instances compared to [93] lies within the expected variance caused by randomization effects. Note that where not otherwise noted, in the default configuration of all local search configurations Bsls, Paig and Pw we will use a seed of value 0 for the RNG.

We compiled a set of in total 16436 benchmarks[1] and included all benchmarks with status *sat* or *unknown* in the QF_BV category of the SMT-LIB [14] benchmark library except those proved by Bb to be unsatisfiable within a time limit of 1200 seconds. We further excluded all benchmarks solved by Boolector via rewriting only. Note that our benchmark set is the same set we already used in [93] and [92]. Previously, all benchmarks in the Sage2 family that used non-SMT-LIBv2 compliant operators had to be explicitly excluded from the set above. However, since the SMT competition 2016, these benchmarks have been removed from SMT-LIB.

All experiments were performed on a cluster with 30 nodes of 2.83 GHz Intel Core 2 Quad machines with 8 GB of memory using Ubuntu 14.04.3 LTS. Each run is limited to use 7 GB of main memory. In terms of runtime we consider CPU time only. In case of a time out or memory out, the time limit is used as runtime. Note that the results in [58] indicate that there still exists a considerable gap between the performance of state-of-the-art bit-blasting and word-level local search. However, the latter significantly outperforms bit-blasting on several instances. We therefore evaluated our local search configurations as in [93] and [92] with regard to an application within a sequential portfolio setting and apply the same time limits, i.e., a limit of 1 and 10 seconds for the local search configurations Bsls, Pw and Paig, and a limit of 1200 seconds for the bit-blasting and sequential portfolio configurations.

Table 7 summarizes the results of configurations Bb, Bsls, Paig and Pw with a time limit of 10 seconds. As illustrated in Figure 8 and 9 in more detail, overall, our word-level propagation strategy Pw clearly outperforms our bit-level propagation strategy Paig and the SLS for SMT approach Bsls. On some benchmarks in the families *sage*, *Sage2* and *stp_samples*, however, in comparison to Bsls (461) and Paig (38) configuration Pw seems to struggle. As an interesting observation, when bit-blasting the benchmarks in question, for the majority of benchmarks more than 50% of the bit-vector expressions contain bits that have been simplified to the Boolean constants $\{0, 1\}$ on the bit-level. Our bit-level strategy Paig operates on the bit-blasted AIG layer where all constant bits are eliminated via rewriting, and therefore always propagates target values that can

[1] All experimental data of this evaluation can be found at `http://fmv.jku.at/fmsd16`.

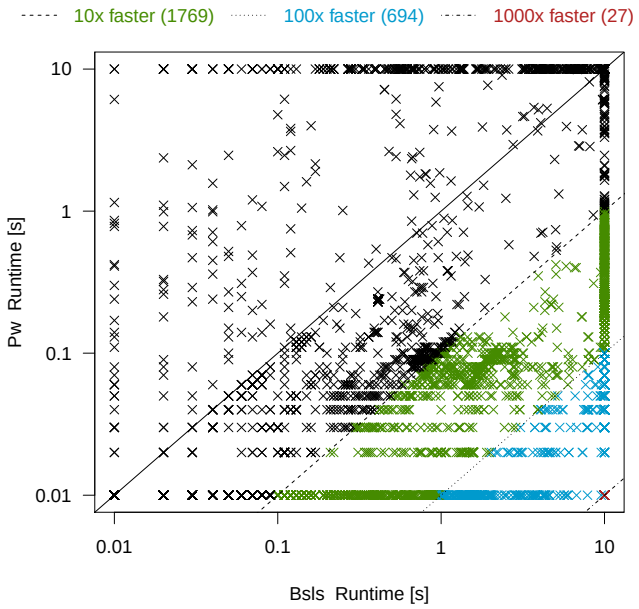


Figure 8: SLS for SMT configuration Bsls versus our propagation strategy Pw with a time limit of 10 seconds. Overall, configuration Pw clearly outperforms configuration Bsls.

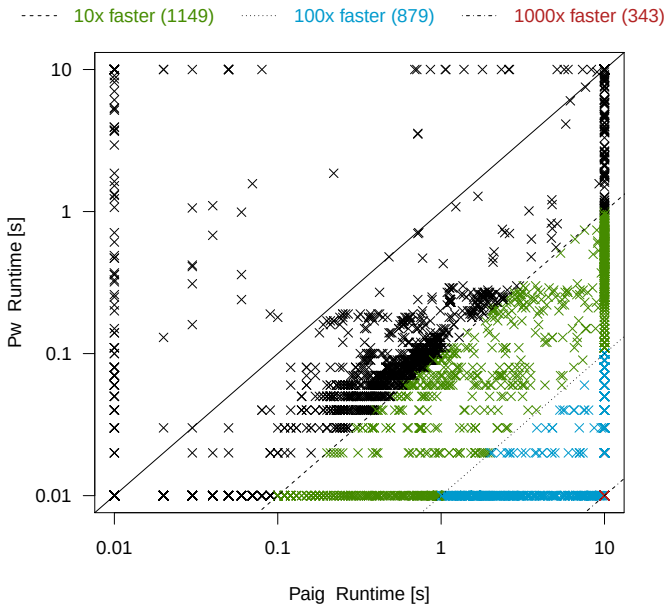


Figure 9: Bit-level propagation strategy Paig versus word-level propagation strategy Pw with a time limit of 10 seconds. Overall, configuration Pw clearly outperforms configuration Paig.

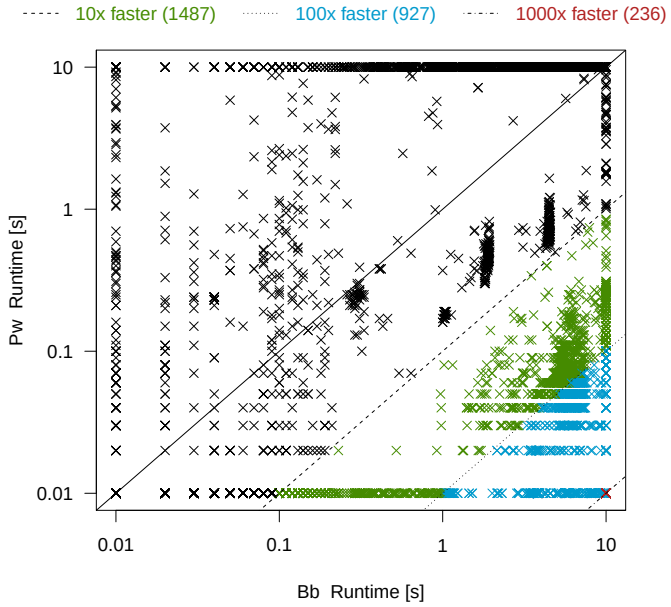


Figure 10: Bit-blasting configuration Bb versus our propagation strategy Pw with a time limit of 10 seconds. Even though Bb solves almost 2000 instances more than Pw, configuration Pw outperforms Bb on 2650 benchmarks in terms of runtime by at least a factor of 10.

be assumed. Our word-level strategy Pw, however, does not know which bits can be simplified to constant bits and may therefore determine and propagate target values that can never be assumed. Configuration Bsls, on the other hand, also does not have any explicit information on constant bits but considers them implicitly when exploring the neighborhood prior to performing a move since any neighbor with constant bits not matching their value will not result in score improvement. We leave the exploration of introducing knowledge on constant bits into our word-level propagation strategy to future work.

Figure 10 shows the performance of our propagation-based configuration Pw compared to our bit-blasting configuration Bb with a time limit of 10 seconds. Even though there exists a considerable gap in the number of solved instances (within 10 seconds, Bb solves almost 2000 instances more than Pw), on more than 2650 benchmarks, configuration Pw outperforms configuration Bb by at least a factor of 10. These results suggest a combination of both configurations within a sequential portfolio setting [104], where our propagation-based strategy is run for a certain amount of time prior to invoking the bit-blasting engine. In practice, however, the number of propagation steps performed is a more reliable metric than the actual runtime of Pw within a sequential portfolio setting. In the following, we distinguish two sequential portfolio configurations.

(1) **Bb+Pw-virtual-Xs** A *virtual* sequential portfolio combination of Pw and Bb, where we assume that Pw is run exactly X seconds prior to invoking Bb.

(2) **Bb+Pw-X** The sequential portfolio combination of Pw and Bb as implemented in Boolector, where configuration Pw is run with a limit of X propagation steps prior to invoking Bb. Note that this configuration won the QF_BV division of the main track of the SMT competition 2016 with X=1000=1k.

Figure 11 illustrates the performance of a virtual sequential portfolio combination Bb+Pw-virtual-1s in comparison to the bit-blasting configuration Bb with a time limit of 1200 seconds, where we assume that configuration Pw is run for one second before falling back to the bit-blasting engine. Overall, configuration Bb+Pw-virtual-1s solves 63 instances more than Bb, and further outperforms Bb in terms of runtime by at least a factor of 10 on almost 2400 benchmarks.

Figure 12 shows the performance of the sequential portfolio combinations Bb+Pw-1k, Bb+Pw-10k, Bb+Pw-50k and Bb+Pw-100k in comparison to configuration Bb with a time limit of 1200 seconds, where Pw is run with a limit of 1 000, 10 000, 50 000 and 100 000 propagation steps before invoking the bit-blasting engine. With a limit of 1k propagation steps, configuration Bb+Pw-1k already solves 41 instances more than Bb. It further outperforms Bb in terms of runtime by at least a factor of 10 on more than 2400 benchmarks. Increasing the propagation step limit for configuration Pw to 10k, 50k and 100k further increases performance in term of runtime, with 2601 (Bb+Pw-10k), 2649 (Bb+Pw-50k) and 2657 (Bb+Pw-100k) instances solved by at least a factor of 10 faster than with configuration Bb. In terms of number of solved instances, configuration Bb+Pw-10k shows the best performance with a plus of 52 instances compared to Bb. Configurations Bb+Pw-50k and Bb+Pw-10k still solve 50 and 45 more instances than Bb, but lose instances compared to Bb+Pw-10k due to the increasing overhead introduced for those instances not solved within the given propagation step limit.

In an additional experiment with configurations Bb+Pw-1k and Bb+Pw-10k, we compiled a set of 21172 unsatisfiable benchmarks containing all QF_BV benchmarks in SMT-LIB with status *unsat* and determined the overhead introduced by Pw. With a total of 1237 seconds for configuration Bb+Pw-1k, the overhead for the unsatisfiable instances is negligible compared to the performance gain of almost 102k seconds on the satisfiable instances. For configuration Bb+Pw-10k, the overhead for the unsatisfiable instances is larger by a factor of 10 (10316 seconds), which is still an order of magnitude less than the performance gain of more than 116k seconds on the satisfiable instances.

Table 8 summarizes the results of configurations Bb, Bb+Pw-virtual-1s and Bb+Pw-10k, and gives a more detailed overview by benchmark family with a time limit of 1200 seconds. As shown in Figure 12, a propagation step limit of 100k (Bb+Pw-100k) almost corresponds to virtually limiting the runtime of Pw

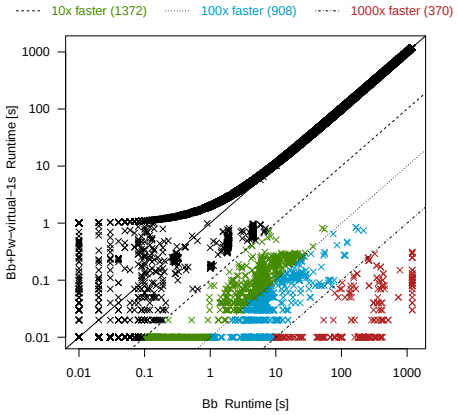


Figure 11: Bb versus our *virtual* sequential portfolio configuration Bb+Pw-virtual-1s with a time limit of 1200 seconds, where Pw is assumed to run with a time limit of 1 second.

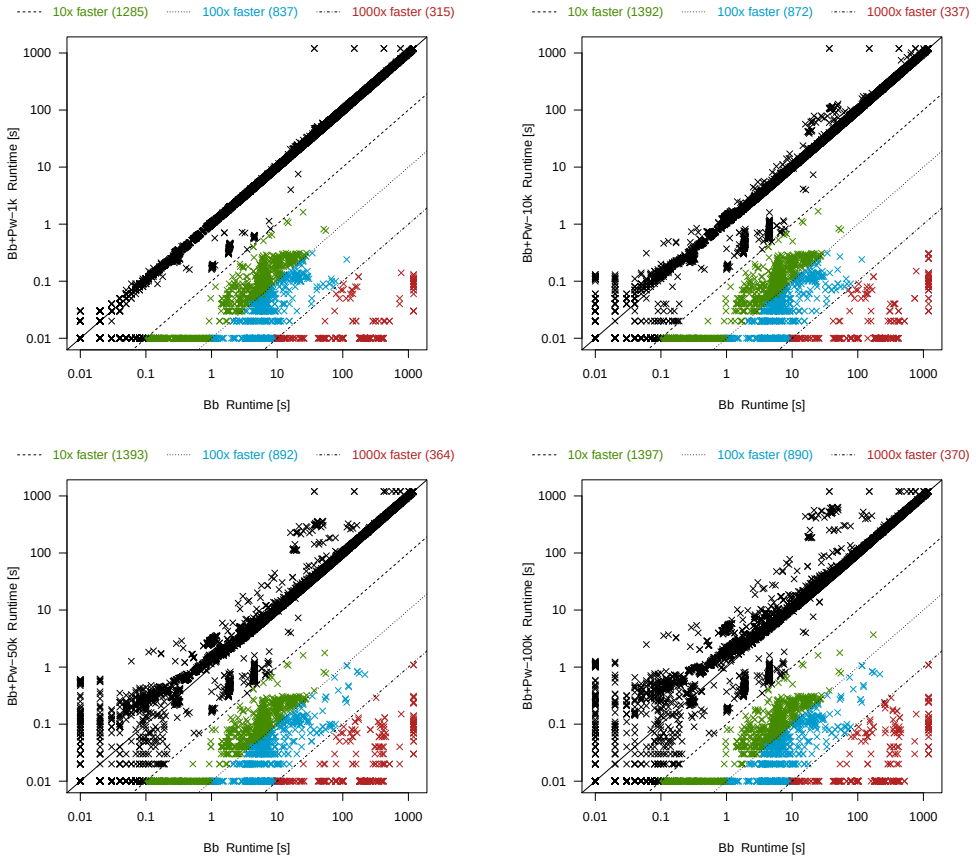


Figure 12: Bb versus our sequential portfolio configurations Bb+Pw-1k, Bb+Pw-10k, Bb+Pw-50k and Bb+Pw-100k with a time limit of 1200 seconds, where Pw is run with a limit of 1k, 10k, 50k and 100k propagation steps.

	Bb		Bb+Pw-10k		Bb+Pw-virtual-1s	
Family	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
asp (376)	289	150600.3	287	155312.1	289	150889.3
bench_ab (223)	223	0.2	223	0.0	223	0.0
bmc (22)	22	63.6	22	83.8	22	71.6
brummayerbiere (26)	17	1759.8	26	46.7	26	97.9
calypto (13)	5	10423.7	5	10420.9	5	10423.5
check2 (1)	1	0.0	1	0.0	1	0.0
crafted (1)	1	0.0	1	0.0	1	0.0
dwp_formulas (103)	103	0.4	103	0.0	103	0.0
fft (19)	5	16924.2	5	16923.7	5	16928.6
float (126)	94	56417.8	94	56276.3	94	55497.3
gulwani (6)	6	47.6	6	48.3	6	53.6
mcm (155)	51	138276.6	51	138368.8	51	138273.6
pspace (21)	21	1964.5	21	1.6	21	1.6
rubik (3)	3	433.9	3	422.6	3	436.9
RWS (20)	18	3635.8	18	3649.8	18	3653.8
sage (6236)	6236	2602.9	6236	2492.5	6236	3622.2
Sage2 (6981)	5898	1853304.4	5940	1752833.2	5949	1738901.0
spear (1675)	1672	16202.9	1675	282.4	1675	278.3
stp (1)	1	18.9	1	20.9	1	19.9
stp_samples (149)	149	4.4	149	10.7	149	75.4
tacas07 (3)	3	11.5	3	6.2	3	6.0
uclid (262)	262	741.3	262	168.7	262	228.4
VS3 (10)	2	9859.6	2	9859.8	2	9861.6
wienand (4)	0	4800.0	0	4800.0	0	4800.0
total (16436)	**15082**	**2268094.4**	**15134**	**2152029.2**	**15145**	**2134120.5**

Table 8: Configurations Bb, Bb+Pw-10k and Bb+Pw-virtual-1s with a time limit of 1200 seconds grouped by benchmark families. Configuration Bb+Pw-virtual-1s is a virtual configuration and combines Bb and Pw in a sequential portfolio with a time limit of 1 second for Pw. Configuration Bb+Pw-10k implements a combination of Bb and Pw within a sequential portfolio where Pw is run with a limit of 10k propagation steps prior to invoking Bb.

to 1 second (Bb+Pw-virtual-1s), in particular when considering the number of instances solved by at least a factor of 10 faster than Bb. A propagation limit of 10 000 (Bb+Pw-10k), however, yields the best results in terms of number of solved instances and the overall runtime.

In order to determine robustness of our propagation-based strategy with respect to randomization effects, and in particular in comparison to the SLS for SMT approach in [58], we ran an additional batch of 20 runs of each Pw, Paig and Bsls with different seeds for the RNG and a time limit of 10 seconds. Figure 13 shows the overall results in terms of number of solved instances and runtime as box-and-whiskers plots with the results of the default configurations (using a seed value of 0) indicated with a red diamond. Over all 21 runs of Pw with different seeds, both the inter-quartile range (IQR) in the number of solved instances (27 instances), i.e., the distance between the lower quartile (8099 instances) and the upper quartile (8126 instances), and the standard deviation (17.9 instances) is less than half of the IQR (60 instances) and the standard deviation (44.9 instances) over all 21 runs of Bsls. In comparison to Paig, the IQR is almost the same, however, the standard deviation of Paig (62.6 instances) is more than triple of the standard deviation of Pw. In terms of runtime, for Pw and Bsls the results are similar, with an IQR of 161.3 seconds and a standard deviation of 149.8 seconds for configuration Pw, and an IQR of 312.5 seconds and a standard deviation of 297.8 seconds for configuration Bsls. Configuration Paig, however, performs considerable worse than the two other configurations, with an IQR of 1761.8 seconds and a standard deviation of 406.4 seconds. These results suggest that compared to Paig, both Pw and Bsls profit from directly working on the word-level. Our propagation-based strategy Pw, however, is indeed more robust with respect to randomization effects than the SLS for SMT approach of [58].

The default configuration of Pw prioritizes inverse values over consistent values during backtracing with a probability of 99:1. Decreasing this ratio, i.e., increasing the probability to choose consistent values over inverse values, increases the level of non-determinism of our backtracing algorithm. Figure 14 illustrates the influence of decreasing the level of non-determinism during value selection on randomization effects in terms of the number of solved instances over 21 runs with different seeds and a time limit of 10 seconds. The default ratio of 99:1 has a standard deviation of 17.9 instances. As might be expected, when introducing higher levels of non-determinism by decreasing the ratio of inverse to consistent values to 50:50 and 0:100 (consistent values only), the standard deviation increases to 23.5 and 38.9 instances. When decreasing the level of non-determinism by increasing the ratio to 100:0 (inverse values only), the standard deviation drops to 14.1 instances. Overall, a higher probability to choose inverse over consistent values also increases performance. However, as shown in Section 4, using inverse values only (ratio 100:0) is incomplete.

In terms of path selection, not prioritizing inputs but choosing randomly corresponds to a maximum level of non-determinism. Prioritizing controlling inputs for Boolean operators already decreases non-determinism during path selection.

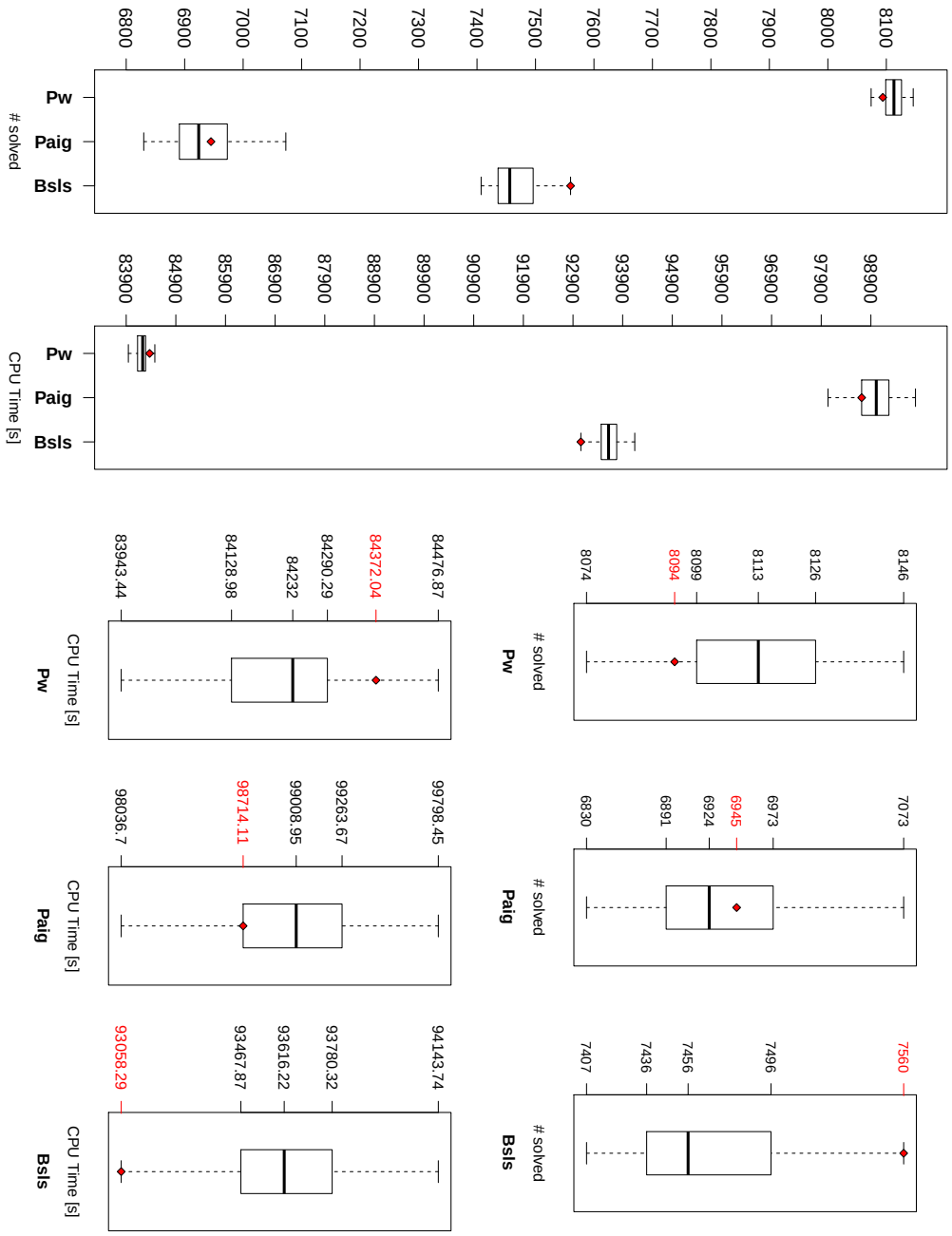


Figure 13: Number of solved instances, run time and randomization effects over 21 runs of configurations Pw, Paig and Bsls with different seeds and a time limit of 10 seconds. Overall, our propagation-based strategy Pw is more robust with respect to randomization effects than the SLS for SMT configuration Bsls.

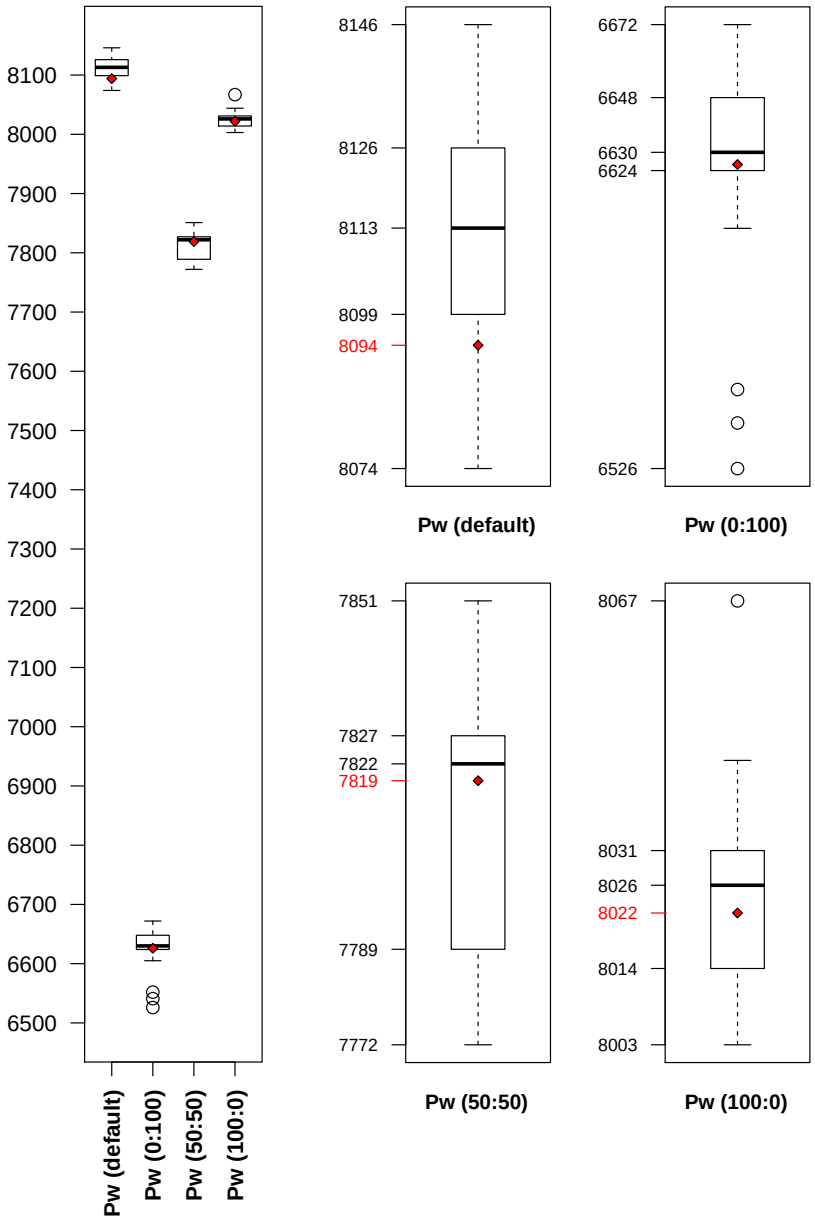


Figure 14: Number of solved instances and randomization effects over 21 runs of configuration Pw with different seeds and different levels of non-determinism during value selection and a time limit of 10 seconds. Configuration Pw (default) prioritizes inverse values over consistent values during backtracing with a probability of 99:1 (default), configuration Pw (0:100) selects consistent values only, configuration Pw (50:50) selects consistent and inverse values with the same probability, and configuration Pw (100:0) selects inverse values only.

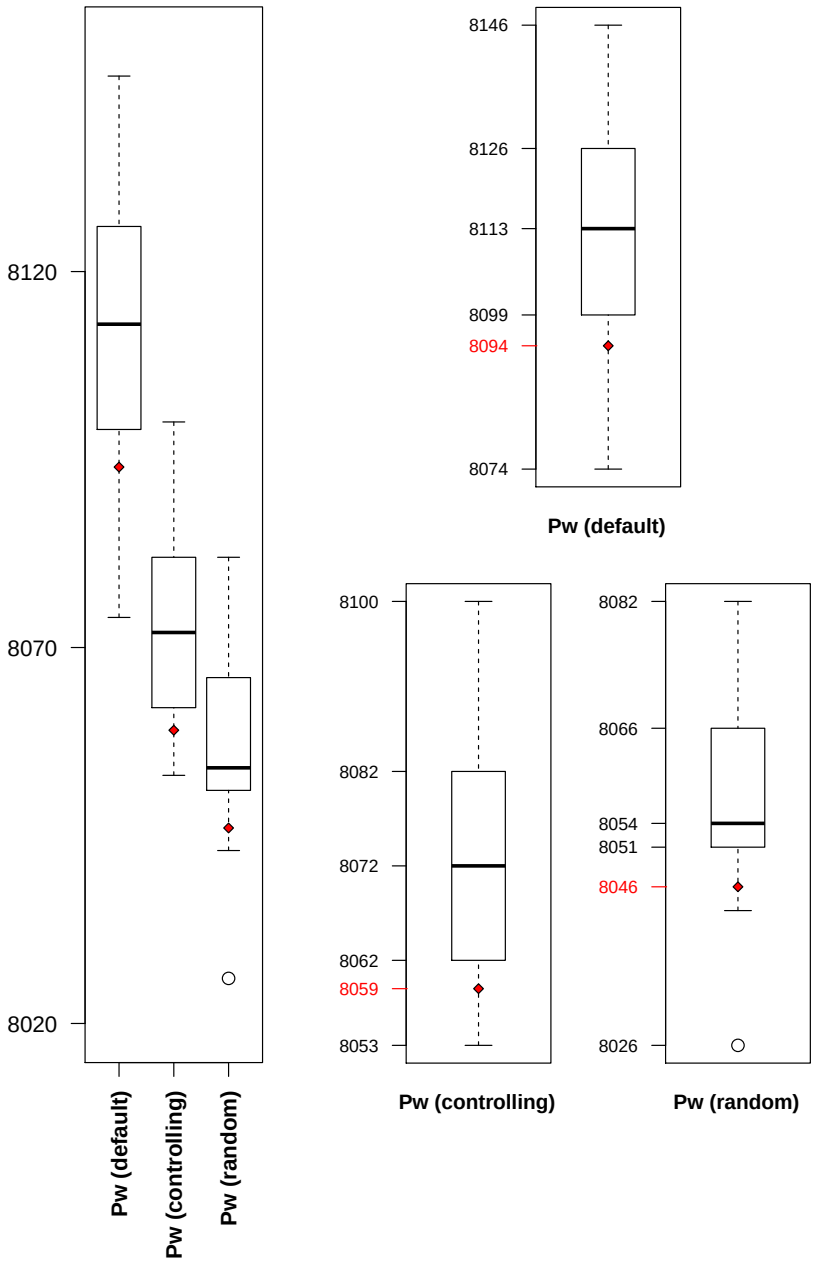


Figure 15: Number of solved instances and randomization effects over 21 runs of configuration Pw with different seeds and different path selection strategies and a time limit of 10 seconds. The default configuration prioritizes essential inputs when down-propagating assignments, whereas configuration Pw (controlling) only prioritizes controlling inputs of Boolean operators. Configuration Pw (random) does not prioritize inputs but chooses randomly.

However, utilizing essential inputs for all word-level operators decreases non-determinism even further. Figure 15 shows the influence of decreasing the level of non-determinism during path selection in terms of the number of solved instances over 21 runs with different seeds and a time limit of 10 seconds. The default configuration of Pw prioritizes essential inputs for all word-level operators, whereas configuration Pw (controlling) only utilizes controlling inputs for Boolean operators. Configuration Pw (random) does not prioritize inputs but chooses randomly. As expected, prioritizing essential inputs for all word-level operators yields the best results. Utilizing only controlling inputs of Boolean operators already decreases performance, and not prioritizing inputs but choosing randomly decreases performance even further.

6 Conclusion

In this paper, we presented our complete propagation-based local search strategy for the theory of quantifier-free fixed-size bit-vectors, which we previously presented in [92], in more detail.

We defined a complete set of rules for determining backtracing values when propagating assignments towards the primary inputs and provided extensive examples to illustrate the core concepts of our approach. We further provided a more extensive experimental evaluation, including an analysis of randomization effects caused by using different seeds for the random number generator. Motivated by the experimental results in [92], which showed the potential of a sequential portfolio combination of our propagation-based strategy and a state-of-the-art bit-blasting approach, we implemented this combination in our SMT solver Boolector. Our results confirm a considerable gain in performance.

Our procedure was evaluated on problems in the theory of quantifier-free bit-vectors in SMT. However, it is not restricted to bit-vector logics. Applying our strategy to other logics is probably the most intriguing direction for future work.

Further, extending our techniques by introducing strategies for conflict detection and resolution during backtracing as well as lemma generation in order to obtain an algorithm that is able to also prove unsatisfiability is another challenge for future work. A possible direction would be incorporating techniques from the MCSat for bit-vectors approach presented in [106].

Finally, we would like to thank Andreas Fröhlich for helpful comments, and Holger Hoos for fruitful discussions on the relation between non-deterministic completeness and the notion of probabilistically approximately complete (PAC).

C

Paper C

Turbo-Charging Lemmas on Demand with Don't Care Reasoning

Published In Proceedings of the 14th International Conference on Formal Methods in Computer Aided Design (FMCAD 2014), pages 179–186, Lausanne, Switzerland, 2014.

Authors Aina Niemetz, Mathias Preiner and Armin Biere.

Fixes Line 15 in Figure 5, subfigure references in caption of Figure 7.

Abstract Lemmas on demand is an abstraction/refinement technique for procedures deciding Satisfiability Modulo Theories (SMT), which iteratively refines full candidate models of the formula abstraction until convergence. In this paper, we introduce a dual propagation-based technique for optimizing lemmas on demand by extracting partial candidate models via don't care reasoning on full candidate models. Further, we compare our approach to a justification-based approach similar to techniques employed in the context of model checking. We implemented both optimizations in our SMT solver Boolector and provide an extensive experimental evaluation, which shows that by enhancing lemmas on demand with don't care reasoning, the number of lemmas generated, and consequently the solver runtime, is reduced considerably.

1 Introduction

Procedures for deciding satisfiability of first-order formulas w.r.t. first-order theories, also known as Satisfiability Modulo Theories (SMT), are usually divided into so-called *eager* and *lazy* approaches. Eager SMT approaches eagerly encode an SMT formula into an equisatisfiable Boolean formula, which then serves as input for a SAT solver. Lazy approaches, on the other hand, are generally based on a tight integration of a SAT solver and one or more theory solvers. The SAT solver typically enumerates Boolean truth assignments satisfying a Boolean abstraction of the input formula, whereas the theory solver(s) not only check if those assignments are consistent w.r.t. the first-order theorie(s), but guide the SAT solver through its search.

The majority of state-of-the-art SMT solvers employ lazy SMT approaches, where the *lemmas on demand* procedure as introduced for the extensional theory of arrays in [33] is one extreme variant thereof [99]. The core idea of lemmas on demand is similar to the Counterexample-Guided Abstraction Refinement (CEGAR) approach introduced in [43] and goes back to [51], while at the same time, a related technique was proposed in the context of bounded model checking, where all-different constraints are lazily encoded over bit vectors (see also [24]). Recently, in [96] we introduced a generalization of the lemmas on demand decision procedure in [33] to lazily handle lambda terms. Similar to other lazy SMT approaches, lemmas on demand as in [33, 96] enumerates truth assignments (so-called *candidate models*) of the bit vector abstraction of the (preprocessed) input formula and iteratively refines those assignments with lemmas until convergence. Each of these candidate models is a full truth assignment of the formula abstraction, which subsequently needs to be checked for consistency w.r.t. the theory of bit vectors with arrays. A full candidate model, however, includes parts of the formula abstraction irrelevant to its satisfiability under the current assignment and might therefore be over-determined.

In this paper we aim at exploiting *a posteriori observability don't cares*, i.e., parts of the formula abstraction irrelevant under the current assignment. We show that don't care reasoning on full candidate models to extract partial candidate models subsequently reduces the cost for consistency checking by focusing on the relevant parts of the formula, only. Motivated by *dual propagation* techniques in the context of quantified boolean formulas (QBF) [64, 65], we propose an optimization of the lemmas on demand procedure in [96] and compare our approach to a technique based on *justification* heuristics in ATPG [87]. We implemented both techniques in our SMT solver Boolector and analyse the results in comparison to the version of Boolector that won the QF_AUFBV track of the SMT competition 2012.

Note that in this paper, our justification-based approach mainly serves as a basis for comparison to our dual propagation-based approach. In the context of SMT, Barrett and Donham [12] and De Moura and Bjørner [48] applied justification-based techniques to prune the search space of DPLL(T). In the

context of model checking, justification-based techniques have been previously employed to identify a posteriori observability don't cares. Bingham and Hu [26], e.g., prune the search space of their simulation-based bounded model checking engine by means of a justification-based generalization mechanism (*skip cubes*) similar to learning and non-chronological backtracking of conventional SAT procedures. Eén et al. [54] employ a related approach when generalizing proof obligations by *ternary simulation* for property directed reachability (PDR), whereas Chockler et al. [41] use a variant of offline dual propagation for SAT. The verification tool Reveal [2,3], on the other hand, employs a CEGAR approach for model checking complex hardware designs and generalizes candidate counter examples by justification techniques similar to our justification-based method. Their (and our) justification-based approach, however, is only applicable on structural (non-clausal) problems. In contrast, our dual propagation-based approach generalizes full candidate models by exploiting the duality of the Boolean layer of the input formula and is not restricted to structural formula abstractions.

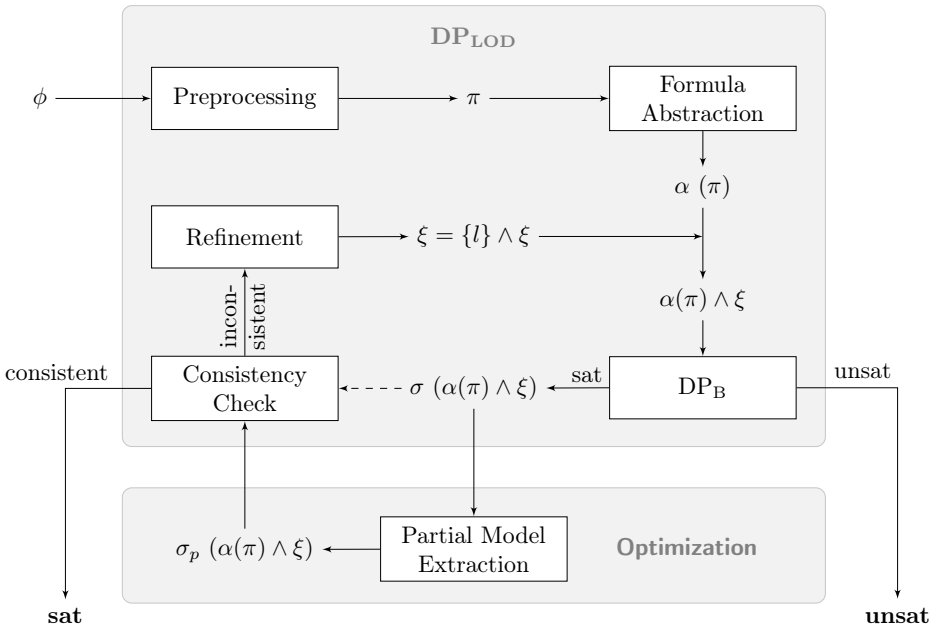


Figure 1: The workflow of the *lemmas on demand* decision procedure $\mathrm{DP}_{\mathrm{LODopt}}$ in Boolector. The original procedure $\mathrm{DP}_{\mathrm{LOD}}$ (indicated by the dashed line) works on full candidate models, whereas the optimized procedure $\mathrm{DP}_{\mathrm{LODopt}}$ extracts partial candidate models prior to consistency checking.

2 Lemmas on Demand at a Glance

The *lemmas on demand* decision procedure as implemented in Boolector is an iterative abstraction/refinement approach for the quantifier-free theory of fixed-sized bit vectors and arrays. Figure 1 gives a high-level view of the procedure and introduces both the original, unoptimized approach $\mathrm{DP}_{\mathrm{LOD}}$ and our optimized approach $\mathrm{DP}_{\mathrm{LODopt}}$ as follows.

Given a formula ϕ, $\mathrm{DP}_{\mathrm{LOD}}$ uses a *bit vector* skeleton of the preprocessed formula π as formula abstraction $\alpha(\pi)$. In each iteration, an underlying decision procedure DP_{B} determines the satisfiability of the refined formula abstraction $\Gamma \equiv \alpha(\pi) \wedge \xi$ by encoding Γ to SAT and determining its satisfiability by means of a SAT solver. Note that initially, formula refinement ξ is $\top$. As Γ is an overapproximation of ϕ, $\mathrm{DP}_{\mathrm{LOD}}$ immediately concludes with *unsat* if Γ is unsatisfiable. If Γ is satisfiable, the current (full) candidate model $\sigma(\alpha(\pi) \wedge \xi)$ is checked for consistency w.r.t. the preprocessed input formula π. If $\sigma(\alpha(\pi) \wedge \xi)$ is consistent, $\mathrm{DP}_{\mathrm{LOD}}$ immediately concludes with *sat*. Otherwise, $\sigma(\alpha(\pi) \wedge \xi)$ is *spurious* and a lemma l is added to formula refinement ξ.

As indicated in Figure 1, $\mathrm{DP}_{\mathrm{LOD}}$ iteratively refines $\alpha(\pi)$ by consistency checking *full* candidate models, which usually include parts of the bit vector skeleton irrelevant to its satisfiability under the current assignment. In the following section, we will introduce an optimization to extract a partial candidate model $\sigma_p(\alpha(\pi) \wedge \xi)$ from the full candidate model $\sigma(\alpha(\pi) \wedge \xi)$ in order to guide the consistency check towards the relevant parts of $\alpha(\pi)$ only.

3 Partial Model Extraction

In terms of runtime, abstraction refinement usually is the most costly part of the lemmas on demand procedure $\mathrm{DP}_{\mathrm{LOD}}$, where cost generally correlates with the number of lemmas generated. During refinement, procedure DP_{B} (and consequently the call to the underlying SAT solver) constitutes the majority of the overall runtime per iteration, which adds up when a great number of refinement iterations is needed. Hence, optimizing $\mathrm{DP}_{\mathrm{LOD}}$ in terms of runtime directly translates to reducing the number of lemmas generated.

In contrast to other lazy SMT approaches [99], formula abstraction in $\mathrm{DP}_{\mathrm{LOD}}$ does not produce a pure Boolean skeleton, but a bit vector skeleton, where each function application $f(a_0, \ldots, a_n)$ in the preprocessed formula π is mapped to a fresh bit vector variable. Consequently, consistency checking in $\mathrm{DP}_{\mathrm{LOD}}$ is performed on *all* function applications in the bit vector skeleton (for details see [96]). A high level view of the consistency checking algorithm consistent in $\mathrm{DP}_{\mathrm{LOD}}$ is given in Figure 2 and proceeds as follows. Given the refined formula abstraction Γ and the full candidate model σ, *search_initial_applies* collects all function applications in Γ that need to be checked for consistency (line 2) and iteratively checks each APPLY $f(a_0, \ldots, a_n)$ w.r.t. the current assignment σ

```
1  procedure consistent (Γ, σ)
2      S ≔ search_initial_applies (Γ)
3      while S ≠ ∅
4          f(a_0, ..., a_n) ≔ pop (S)
5          consistent ≔ check_consistency (f(a_0, ..., a_n), σ)
6          if not consistent
7              return ⊥
8          S' ≔ search_applies_for_consistency_check (f(a_0, ..., a_n))
9          push (S, s' ∈ S')
10     return ⊤
```

Figure 2: Procedure *consistent* in pseudo-code.

(lines 4-5). If *check_ consistency* encounters an inconsistency, *consistent* immediately returns with $\bot$. Else, *search_ applies_for_ consistency_ check* instantiates function f with arguments $a_0, \ldots, a_n$, which yields term t, and subsequently collects all function applications in formula abstraction $\alpha(t)$ for consistency checking (lines 8-9). If all applies in S have been checked without inconsistencies, procedure *consistent* concludes that current candidate model σ is consistent and returns $\top$.

Consistency checking all function applications in formula abstraction Γ corresponds to checking the *full* candidate model σ for consistency, with the order in which applies are checked as the only way to positively influence the number of refinement iterations (by coincidentally finding lemmas that shortcut the search, early on). Checking the full candidate model, however, is often not required, as only a small subset of the full candidate model is responsible for actually satisfying the formula abstraction. As a consequence, parts of the formula abstraction irrelevant to its satisfiability under the current assignment are checked, which subsequently produces lemmas that do not necessarily prune the search space and therefore mainly cost runtime.

Example 1. As a running example, consider the formula

$$\psi_1 \equiv i \neq k \wedge (f(i) = e \vee f(k) = v) \wedge v = ite(i = j, e, g(j))$$

as given in Figure 3. Its initial formula abstraction $\Gamma_{\psi_1} \equiv \alpha(\psi_1)$ and a (possible) initial full candidate model $\sigma(\Gamma_{\psi_1})$ (indicated in red) is given in Figure 4. In the following, we assume that all variables in ψ_1 are bit vector variables of size 2 and Γ_{ψ_1} is a bit vector skeleton. For the sake of simplicity, we further assume that functions f and g represent uninterpreted functions, i.e., we concentrate on consistency checking of the *full* versus a *partial* candidate model (via procedure *search_ initial_ applies*) and do not bother with details of the internals of the actual consistency check (for details, see [96]). Procedure *search_ initial_ applies*

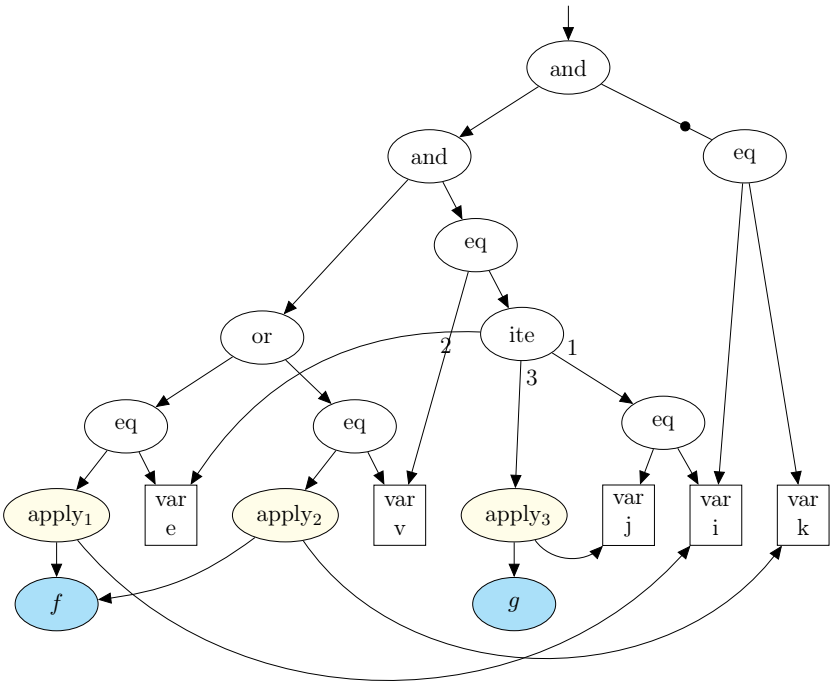


Figure 3: DAG representation of formula ψ_1 (running example).

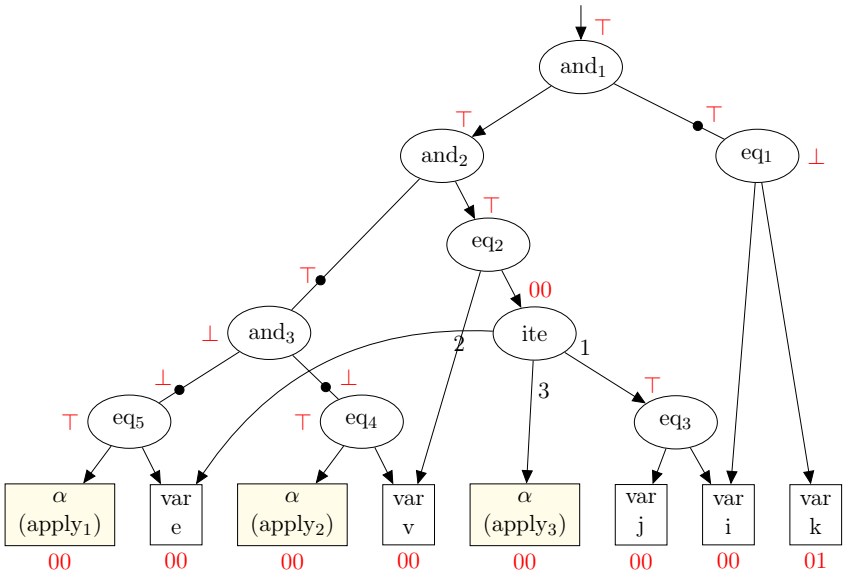


Figure 4: Formula abstraction Γ_{ψ_1} of formula ψ_1 with candidate model $\sigma(\Gamma_{\psi_1})$ indicated in red (running example).

initially collects all function applications in Γ_{ψ_1} (apply_1, apply_2, apply_3) to be checked for consistency. During consistency checking, however, no further applies are identified as required to being checked (procedure *search_ applies_ for_ consistency_ check*) as both f and g do not make subsequent calls to other functions. Note that given $\sigma(\Gamma_{\psi_1})$, instead of checking all applies in ψ_1, either checking apply_1 or apply_2 would be sufficient.

In the following, we consider two techniques for identifying irrelevant parts of the formula abstraction by extracting partial candidate models, which subsequently reduces the number of refinement iterations, and therefore, the overall runtime of the lemmas on demand procedure.

3.1 Justification-Based Partial Model Extraction

In the context of ATPG [87], sets of don't care conditions are usually divided into *observability don't cares (ODC)* and *controllability don't cares (CDC)*. The former denotes lines that do not influence the primary outputs (independent from the current assignment to the primary inputs), and the latter identifies line values that can not be justified and are therefore illegal under any assignment to the primary inputs. Given a concrete assignment to the primary inputs, however, we can determine what we call *a posteriori* observability don't cares, i.e., lines that do not influence the output of a gate under its current assignment. In the context of model checking, such *a posteriori* ODC have already been exploited by Bingham and Hu [26], Eén et al. [54], and Andraus et al. [2, 3].

In this section, we introduce a technique similar to [2, 3] and extract partial candidate models by identifying parts of the formula abstraction Γ that are irrelevant to its satisfiability under the current assignment σ. As indicated above, this directly translates to collecting and checking function applications in relevant parts of Γ only. In the following, we assume that Γ is represented as a directed acyclic graph (DAG) with exactly one root, where all Boolean operations are expressed by means of NOT and (two-input) AND gates. In place of procedure *search_ initial_ applies*, we introduce *search_ initial_ applies*$_{just}$ (Figure 5), which collects function applications while traversing all relevant paths in Γ as follows.

Given Γ and a full candidate model σ, starting from the root, *search_ initial_ applies*$_{just}$ iteratively traverses Γ towards its primary inputs (bit vector variables and function applications) in depth first search (DFS) order. That is, initially, $\text{root}(\Gamma)$ is pushed onto stack X (line 2) and for each node $x \in X$ we determine the paths to be skipped as follows. If a node x is an AND node and its output is assigned to $\bot$, we follow (one of) its controlling input(s), i.e., one of its inputs with controlling value ($\bot$ for an AND) [87], and skip the other (lines 7-14). Similarly, if x is an IF-THEN-ELSE (ITE) node and its condition is assigned to $\top$ (resp. $\bot$), we follow both its condition and its *then* (resp. *else*) branch (lines 15-20). In any other case where x is not an APPLY node, we follow all inputs of node x (line 22). However, if x is an APPLY node, we collect x (line 6)

```
1  procedure search_initial_applies_just (Γ, σ)
2      S ≔ ∅, X ≔ {root(Γ)}
3      while X ≠ ∅
4          x ≔ pop (X)
5          if is_apply (x)
6              push (S, x)
7          else if is_and (x) and σ (x) = ⊥
8              l ≔ left_input (x), r ≔ right_input (x)
9              if is_controlling (l) and is_controlling (r)
10                 push (X, choose (l, r))
11             else if is_controlling (l)
12                 push (X, l)
13             else
14                 push (X, r)
14         else if is_ite (x)
15             push (X, condition (x))
16             if σ (condition (x)) = ⊤
17                 push (X, then (x))
18             else
19                 push (X, else (x))
20         else
21             push (X, i ∈ inputs (x))
22     return S
```

Figure 5: Procedure *search_initial_applies*$_{just}$ in pseudo-code.

and cut off the traversal, as function applications are treated as fresh bit vector variables in the formula abstraction.

Note that in the case that *both* inputs of an AND node are controlling, we can skip *either* one of them (lines 9-10). Hence, we choose to follow the input with minimum cost in terms of consistency checking, where the cost of a node x is defined as the minimum number of (unique) applies along a path from x to the primary inputs in the preprocessed formula π. Similar as controllability measures in ATPG [87], we recursively define a cost function $\mathrm{cost}(x)$ as follows.

$$\mathrm{cost}(x) = \begin{cases} 0 & \text{if } \mathrm{is\_var}(x) \\ \min(\mathrm{cost}(i) \mid i \in \mathrm{inputs}(x)) & \text{if } \mathrm{is\_and}(x) \\ \mathrm{sum}(\mathrm{cost}(i) \mid i \in \mathrm{inputs}(x)) + 1 & \text{if } \mathrm{is\_apply}(x) \\ \mathrm{sum}(\mathrm{cost}(i) \mid i \in \mathrm{inputs}(x)) & \text{otherwise} \end{cases}$$

Given formula π, a bit vector variable is a primary input, hence its cost is defined as 0. Function applications, on the other hand, are not primary inputs

but define the cost of a path from input x to the primary inputs. Hence, the cost of an APPLY is defined as the sum of the costs of its inputs increased by one. In case of an AND node, we want to choose the input with minimum cost if both inputs are controlling, hence cost is defined as the minimum cost of its inputs. In any other case, all input paths have to be followed and $\text{cost}(x)$ is defined as the sum of the costs of all inputs of x.

Example 2. Consider formula ψ_1, formula abstraction Γ_{ψ_1}, and a full candidate model $\sigma(\Gamma_{\psi_1})$ as given in Example 1. Starting from the root (and_1), procedure $search\_initial\_applies_{just}$ traverses Γ_{ψ_1} in DFS order while identifying (and skipping) all paths irrelevant w.r.t. assignment $\sigma(\Gamma_{\psi_1})$. Note that in Figure 3 and 4, inverted nodes are indicated by black dots. In the following, however, we will interpret an inverted node as two distinct nodes (with resp. distinct assignments), i.e., $\neg\text{and}_3$ with $\sigma(\neg\text{and}_3) = \top$ in Figure 4, for example, is treated as a NOT (assigned to $\top$) in front of an AND (assigned to $\bot$). Starting with root and_1, which is assigned to $\top$, neither of its inputs may be skipped and we first travel down towards eq_1, whose inputs are both bit vector variables. Hence, we immediately continue with and_2 (also assigned to $\top$) and follow its input eq_2, where we encounter an ite with its condition assigned to $\top$. We skip the *else* branch, no APPLY is collected, and we continue down the input path leading to and_3, which is assigned to $\bot$. Both inputs of and_3 are controlling (i.e., assigned to $\bot$), hence we choose one of them heuristically. The minimum cost for both paths, however, is 0 (as the body of function f does not contain any further applies), hence we may choose either. We decide on the path to apply_1 and conclude with $S = \{\text{apply}_1\}$, which corresponds to the partial model to be subsequently checked for consistency.

3.2 Dual Propagation-Based Partial Model Extraction

Exploiting the duality of QBF by propagating a dual set of values through a QBF ϕ and its negation $\neg\phi$, also referred to as *dual propagation*, has successfully been employed in [64] to significantly prune, and therefore speed up the search in circuit-based QBF solvers. The core idea of *dual propagation*, however, is neither restricted to circuit-based representations [65] nor to QBF and is based on the fact that assignments satisfying an input formula ϕ (the *primal* channel), falsify its negation $\neg\phi$ (the *dual* channel) and vice versa. Given a Boolean formula $\psi_2 \equiv (a \wedge b) \vee (c \wedge d)$, for example, assignment $\{\sigma(a) = \top, \sigma(b) = \top, \sigma(c) = \top, \sigma(d) = \top\}$ satisfies ψ_2, but falsifies its negation $\neg\psi_2 \equiv (\neg a \vee \neg b) \wedge (\neg c \vee \neg d)$.

The duality of formula ψ_2, however, can be further exploited. Assume, for example, that given ψ_2 and $\sigma(\psi_2)$ as above, we fix the values of all input variables assigned in $\sigma(\psi_2)$ by making assumptions $\{a = \top, b = \top, c = \top, d = \top\}$ to a SAT solver maintaining its negation $\neg\psi_2$. All assumptions inconsistent with $\neg\psi_2$, also called *failed assumptions* [55], identify all input assignments sufficient to falsify $\neg\psi_2$, hence sufficient to satisfy ψ_2. This set of failed assumptions, for example $\{a = \top, b = \top\}$, therefore represents a *partial model* satisfying ψ_2.

```
1  procedure search_initial_applies_dp (Γ, σ)
2      S := ∅, A := ∅
3      assume (dual_solver, ¬Γ)
4      X := collect_primary_inputs (Γ)
5      for x in X
6          a := x = σ (x), A := A ∪ a
7          assume (dual_solver, a)
8      res := DP_B (dual_solver)
9      assert res = UNSAT
10     for a in A
11         (x, σ (x)) := a
12         if is_failed (a) and is_apply (x)
13             push (S, x)
14     return S
```

Figure 6: Procedure *search_initial_applies*$_{dp}$ in pseudo-code. Solver instance *dual_solver* simulates the dual channel and is maintained globally.

Note that our approach does not require a structural SAT solver—structural don't care reasoning is simulated via the dual solver, which maintains $\neg\psi_2$ in CNF. Consequently, given a CNF representation of ψ_2 (where structural information of ψ_2 is essentially lost), we extract a partial model (disregarding structural don't cares w.r.t. assignment σ) that satisfies ψ_2 but not necessarily its encoding to CNF. Consider, for example, the Tseitin encoding $\text{CNF}(\psi_2) \equiv (\neg o \vee x \vee y) \wedge (\neg x \vee o) \wedge (\neg y \vee o) \wedge (\neg x \vee a) \wedge (\neg x \vee b) \wedge (\neg a \vee \neg b \vee x) \wedge (\neg y \vee c) \wedge (\neg y \vee d) \wedge (\neg c \vee \neg d \vee y)$. Our previous partial model $\{a = \top, b = \top\}$ satisfies ψ_2 (and therefore identifies those parts of ψ_2 relevant to its satisfiability) but does not satisfy all clauses in $\text{CNF}(\psi_2)$. This is in contrast to other partial model extraction techniques based on iterative removal of unnecessary assignments on the CNF level (e.g. [52]), which do not enable structural don't care reasoning and therefore need to satisfy all clauses in $\text{CNF}(\psi_2)$.

In this section, we lift the approach sketched above to the word level by means of a dual SMT solver and introduce a technique to extract partial candidate models via dual propagation-based don't care reasoning. Given a formula abstraction $\Gamma \equiv \alpha(\pi) \wedge \xi$, we use a single *dual* solver instance to maintain $\neg\Gamma$ over all refinement iterations in combination with the *primal* (or *main*) solver. However, since in each iteration i a new lemma l_i is added to $\xi \equiv l_1 \wedge ... \wedge l_{i-1}$, we set up the dual solver to maintain $\neg\Gamma \equiv \neg(\alpha(\pi) \wedge l_1 \wedge ... \wedge l_{i-1} \wedge l_i)$ as assumption rather than assertion. As illustrated in Figure 6, we introduce *search_initial_applies*$_{dp}$ in place of procedure *search_initial_applies* as follows.

Given Γ and a full candidate model σ, procedure *search_initial_applies*$_{dp}$ initializes the dual solver by assuming $\neg\Gamma$ (line 3). The value of all primary inputs in $\neg\Gamma$ is then fixed by making assumptions of the form $x = \sigma(x)$, where x is either a bit vector variable or an abstracted function application, and $\sigma(x)$ is its assignment in the current full candidate model σ (lines 4-7). Candidate model σ represents a satisfying assignment for Γ, hence decision procedure $\mathrm{DP_B}$ must conclude that assuming σ, $\neg\Gamma$ is unsatisfiable (lines 8-9). The resulting set of failed assumptions identifies all relevant parts of Γ w.r.t. assignment σ, and all function applications in the set of failed assumptions are subsequently collected for consistency checking (lines 10-13).

Example 3. Again, consider formula ψ_1, its initial formula abstraction $\Gamma_{\psi_1} \equiv \alpha(\psi_1)$, and a (possible) full candidate model $\sigma(\psi_1)$ as given in Example 1. Procedure *search_initial_applies*$_{dp}$ initializes the dual solver by assuming $\neg\Gamma_{\psi_1} \equiv \neg(i \neq k \wedge (\alpha(\mathrm{apply}_1) = e \vee \alpha(\mathrm{apply}_2) = v) \wedge v = ite(i = j, e, \alpha(\mathrm{apply\_3})))$, and subsequently collects all bit vector variables i, j, k, e, v and abstracted function applications $\alpha(\mathrm{apply\_1})$, $\alpha(\mathrm{apply}_2)$, $\alpha(\mathrm{apply}_3)$ in Γ_{ψ_1} onto stack X. All primary inputs $x \in X$ are then fixed by making assumptions $\{i = 00, j = 00, k = 01, e = 00, v = 00, \alpha(\mathrm{apply}_1) = 00, \alpha(\mathrm{apply}_2) = 00, \alpha(\mathrm{apply}_3) = 00\}$ to the dual SMT solver instance, which concludes that under the current set of assumptions, $\neg\Gamma_{\psi_1}$ is unsatisfiable. Assumption $\alpha(\mathrm{apply}_1) = 00$ is identified as failed assumption and we conclude with $S = \{\mathrm{apply}_1\}$ to be subsequently checked for consistency.

Note that in a sense, our dual propagation-based approach as discussed above simulates dual propagation as introduced in the context of QBF [64, 65] rather than literally lifting it to bit vectors with arrays. Dual propagation as in [64, 65] is done *eagerly* by means of one single solver instance maintaining a primal and a dual channel without additional overhead. Primary inputs are shared between both channels, which enables symmetric propagation between the primal and dual channel and allows to detect partial models early—even before a full assignment has been generated. In our approach, however, propagation is not interleaved, but consecutive—the primal solver generates a full assignment before the dual solver enables partial model extraction based on the primal full assignment. Further, primary inputs are not physically shared as the dual solver discretely maintains $\neg\phi$ (while mapping primary inputs back to the primal solver and vice versa). Hence we have to simulate shared inputs via fixing input values by means of assumptions to the dual solver, which simply acts as "slave" for partial model extraction to the primal solver. In order to adopt a more eager approach to enable early partial model extraction while reducing the dual solver overhead, interleaved execution between the primal and dual solver similar to "SAT modulo SAT" [17] would be required. Integrating such an interleaved decision process into an existing SMT solver has high potential, however, is rather involved to implement and left to future work.

4 Experimental Evaluation

We implemented justification-based and dual propagation-based partial model extraction in our SMT solver Boolector and provide a comparison of the following four configurations:

(1) **Boolector$_{\mathbf{sc}}$** The version that won the QF_AUFBV track of the SMT competition 2012.

(2) **Boolector$_{\mathbf{ba}}$** Our current base version of Boolector, a slightly optimized version of [96], with partial model extraction disabled.

(3) **Boolector$_{\mathbf{ju}}$** Our base version of Boolector with justification-based partial model extraction enabled.

(4) **Boolector$_{\mathbf{dp}}$** Our base version of Boolector with dual propagation-based partial model extraction enabled.

We compiled two benchmarks sets for our experimental evaluation: (1) *SMT'12* (149 instances), which consists of all non-extensional benchmarks used for the SMT competition 2012 and (2) *Selected* (173 instances), which includes all non-extensional benchmarks from the QF_AUFBV category of SMT-LIB [14] for which Boolector$_{\text{sc}}$ required at least 10 seconds (CPU time) for solving (incl. timeouts and memouts). Note that we had to exclude extensional benchmarks as Boolector$_{\text{ba}}$ and its optimized versions Boolector$_{\text{ju}}$ and Boolector$_{\text{dp}}$ do not yet support extensionality on arrays. Further note that 58 instances of the benchmark set *SMT'12* are included in *Selected*. All experiments were performed on 2.83GHz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 12.04. The memory and time limits for each solver instance were set to 7GB and 1200 seconds, respectively.

4.1 Results Overview

The overall results of all four solver configurations on both benchmark sets *SMT'12* and *Selected* are shown in Table 1, which summarizes the number of solved instances (Solved), timeouts (TO), memouts (MO), total CPU time (Time), and the overhead produced by the dual solver in terms of CPU time (DS). Note that the overhead introduced by our justification-based approach is negligible. Further note that in case of a timeout or memout, a penalty of 1200 seconds was added to the total CPU time. On the *SMT'12* benchmark set, in terms of solved instances, Boolector$_{\text{ba}}$, Boolector$_{\text{ju}}$, and Boolector$_{\text{dp}}$ perform slightly better than Boolector$_{\text{sc}}$. In terms of runtime, however, only Boolector$_{\text{ju}}$ shows a significant improvement (of about 20%), while Boolector$_{\text{dp}}$ appears to even perform worse than Boolector$_{\text{ba}}$, which is mainly due to the runtime overhead introduced by the dual solver. If we disregard this overhead, the overall

	Solver	Solved (sat/unsat)	TO	MO	Time [s]	DS [s]
SMT'12	$\text{Boolector}_{\text{sc}}$	140 (83/57)	9	0	15882	-
	$\text{Boolector}_{\text{ba}}$	141 (83/58)	8	0	19312	-
	$\text{Boolector}_{\text{ju}}$	142 (84/58)	7	0	15709	-
	$\text{Boolector}_{\text{dp}}$	142 (84/58)	7	0	20992	5045
Selected	$\text{Boolector}_{\text{sc}}$	116 (72/44)	50	7	85863	-
	$\text{Boolector}_{\text{ba}}$	121 (76/45)	45	7	76104	-
	$\text{Boolector}_{\text{ju}}$	130 (85/45)	36	7	63202	-
	$\text{Boolector}_{\text{dp}}$	130 (85/45)	36	7	66991	4705

Table 1: Overall results on sets *SMT'12* and *Selected*.

runtime of $\text{Boolector}_{\text{dp}}$ is competitive with the runtime of $\text{Boolector}_{\text{ju}}$. It is conceivable that an eager implementation of dual propagation would perform equally well, i.e., at least as fast as $\text{Boolector}_{\text{dp}}$ without the overhead.

Interestingly, $\text{Boolector}_{\text{sc}}$ clearly outperforms all other three solver configurations on the benchmark family "platania strcmp" (9 instances). $\text{Boolector}_{\text{sc}}$ solved these benchmarks in about 31 seconds, whereas the other solvers required 4416 seconds ($\text{Boolector}_{\text{ba}}$), 2308 seconds ($\text{Boolector}_{\text{ju}}$), and 4527 seconds ($\text{Boolector}_{\text{dp}}$, incl. 2277 seconds dual solver overhead), respectively. The base version $\text{Boolector}_{\text{ba}}$, and consequently both $\text{Boolector}_{\text{ju}}$ and $\text{Boolector}_{\text{dp}}$, obviously struggle on these benchmarks, which needs further investigation.

Note that benchmark set *SMT'12* is not necessarily representative for lemmas on demand in Boolector, as 79 (53%) out of a total of 149 instances are immediately solved by $\text{Boolector}_{\text{sc}}$ without a single refinement iteration. Benchmark set *Selected*, on the other hand, has been compiled based on the runtime performance of the SMT competition 2012 winner $\text{Boolector}_{\text{sc}}$ (incl. timeouts and memouts) and represents a set considered to be "harder" for Boolector. As indicated in Table 1, on set *Selected* both $\text{Boolector}_{\text{ju}}$ and $\text{Boolector}_{\text{dp}}$ clearly outperform their base version $\text{Boolector}_{\text{ba}}$ as well as the competition configuration $\text{Boolector}_{\text{sc}}$. More precisely, both our justification-based and dual propagation-based optimizations considerably reduce the overall runtime while solving 14 (9) additional instances compared to $\text{Boolector}_{\text{sc}}$ ($\text{Boolector}_{\text{ba}}$), where 13 (9) out of 14 (9) are satisfiable instances. Again, $\text{Boolector}_{\text{dp}}$ is slowed down by the dual solver overhead, but still manages to solve as many instances as $\text{Boolector}_{\text{ju}}$. Disregarding the dual solver overhead, $\text{Boolector}_{\text{dp}}$ even outperforms $\text{Boolector}_{\text{ju}}$ in terms of runtime. Note that the dual solver overhead in general correlates with the number of lemmas generated. This is due to the fact that in each refinement

	Solver	Time [s]			Sat [s]			DS Overhead [s]		
		Total	Avg.	Med.	Total	Avg.	Med.	Total	Avg.	Med.
SMT'12	$\text{Boolector}_{\text{sc}}$	4129	29	2	3662	26	0	-	-	-
	$\text{Boolector}_{\text{ba}}$	8564	61	6	7262	52	1	-	-	-
	$\text{Boolector}_{\text{ju}}$	6362	45	4	5226	37	0	-	-	-
	$\text{Boolector}_{\text{dp}}$	10145	72	5	4700	33	0	4109	29	0
Selected	$\text{Boolector}_{\text{sc}}$	15037	133	35	12836	113	34	-	-	-
	$\text{Boolector}_{\text{ba}}$	10001	88	35	8330	73	22	-	-	-
	$\text{Boolector}_{\text{ju}}$	8182	72	29	6639	58	19	-	-	-
	$\text{Boolector}_{\text{dp}}$	10838	95	30	6164	54	15	3036	26	0

	Solver	LOD			Array Model Size		
		Total	Avg.	Med.	Total	Avg.	Med.
SMT'12	$\text{Boolector}_{\text{sc}}$	30741	221	0	184032	2272	20
	$\text{Boolector}_{\text{ba}}$	33013	237	0	33310	411	20
	$\text{Boolector}_{\text{ju}}$	23660	170	0	19751	243	13
	$\text{Boolector}_{\text{dp}}$	33492	240	0	27912	344	12
Selected	$\text{Boolector}_{\text{sc}}$	104646	926	175	512225	7645	1257
	$\text{Boolector}_{\text{ba}}$	31752	280	88	136681	2040	212
	$\text{Boolector}_{\text{ju}}$	28215	249	28	122763	1832	154
	$\text{Boolector}_{\text{dp}}$	24866	220	29	130440	1946	170

Table 2: Results for commonly solved instances on sets *SMT'12* (139 benchmarks, 82 sat, 57 unsat) and *Selected* (113 benchmarks, 70 sat, 43 unsat). Commonly solved satisfiable instances for determining array model size were 81 (out of 82) for *SMT'12* and 67 (out of 70) for *Selected*. Array model size is measured in terms of number of index/value pairs.

iteration a partial candidate model is extracted from the full candidate model, which requires an additional call to the dual solver. On set *Selected*, for 10 out of 130 instances, the dual solver overhead constitutes about 50-70% of the total runtime per instance, whereas for 83 instances it does not exceed 10%.

4.2 Results Commonly Solved Instances

Table 2 summarizes all instances in each benchmark set that could be solved by all four solver configurations and gives an overview of the runtime required for solving (Time), the runtime required by the underlying SAT solver (Sat), the dual solver overhead (DS), the number of lemmas generated (LOD), and the size of the array models for satisfiable instances (Array Model Size). For all four solver configurations, we identified 139 common instances (82 sat, 57 unsat) on benchmark set *SMT'12* and 113 common instances (70 sat, 43 unsat) on benchmark set *Selected*. Array model size is measured in terms of the number of index/value pairs identified by each solver with model generation enabled. However, unlike $\text{Boolector}_{\text{ba}}$ (and consequently $\text{Boolector}_{\text{ju}}$ and $\text{Boolector}_{\text{dp}}$), $\text{Boolector}_{\text{sc}}$ requires additional overhead for model generation, which has a negative impact on the overall number of solved instances. As a consequence, $\text{Boolector}_{\text{sc}}$ effectively "lost" 1 (resp. 3) satisfiable instance(s) on set *SMT'12* (resp. *Selected*). We therefore compiled all columns except column Array Model Size with model generation disabled.

On the 139 common instances in the *SMT'12* benchmark set, $\text{Boolector}_{\text{sc}}$ is still the fastest solver, albeit only due to the "platania strcmp" benchmarks mentioned above—on those nine instances, $\text{Boolector}_{\text{ba}}$, $\text{Boolector}_{\text{ju}}$, and $\text{Boolector}_{\text{dp}}$ spent 50%, 35% and 45% of the overall runtime, respectively. A similar picture emerges when comparing the number of refinement iterations required for these nine instances, which constitutes 59%, 47%, and 60% of the total number of lemmas generated by $\text{Boolector}_{\text{ba}}$, $\text{Boolector}_{\text{ju}}$, and $\text{Boolector}_{\text{dp}}$, respectively. In comparison to the base version $\text{Boolector}_{\text{ba}}$, however, $\text{Boolector}_{\text{dp}}$ shows the most notable improvement (about 26%) in terms of runtime required by the underlying SAT solver on the 139 common instances in *SMT'12*. Disregarding the dual solver overhead, $\text{Boolector}_{\text{dp}}$ even outperforms $\text{Boolector}_{\text{ju}}$ in terms of overall runtime. Interestingly, in terms of the number of lemmas generated, $\text{Boolector}_{\text{dp}}$ requires slightly more lemmas than the base version, which is in stark contrast to $\text{Boolector}_{\text{ju}}$. However, in case of $\text{Boolector}_{\text{dp}}$, this can be contributed to a relative small number of instances. On 14 instances, $\text{Boolector}_{\text{dp}}$ generates 1.5 to 2.6 times more lemmas than $\text{Boolector}_{\text{ba}}$, whereas on all other instances, $\text{Boolector}_{\text{ba}}$ requires considerably more refinement iterations than $\text{Boolector}_{\text{dp}}$. This might indicate that in some cases, $\text{Boolector}_{\text{ba}}$ coincidentally generates lemmas that shortcut the search early on. In terms of array model size, both optimized configurations $\text{Boolector}_{\text{ju}}$ and $\text{Boolector}_{\text{dp}}$ clearly show a reduction in the number of array index/value pairs compared to the base version $\text{Boolector}_{\text{ba}}$.

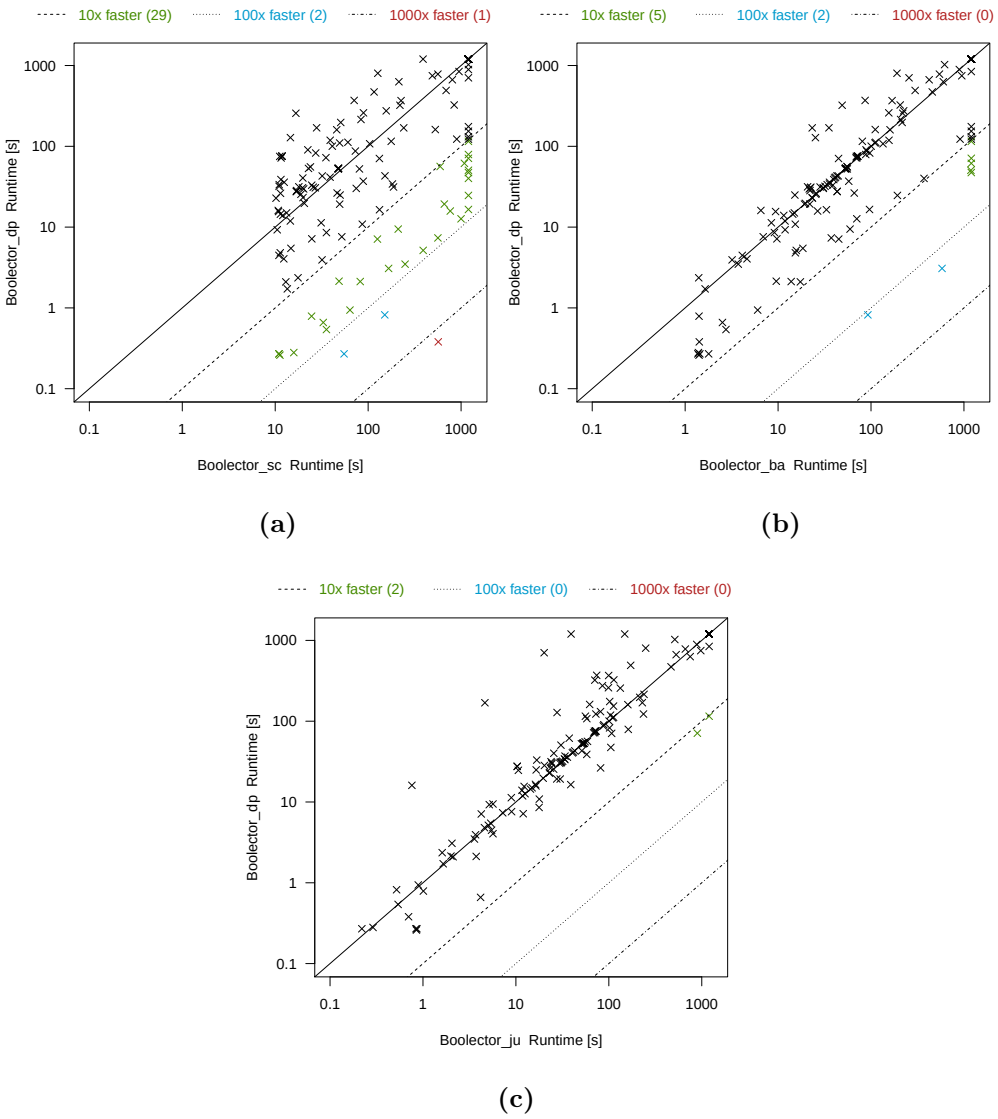


Figure 7: Runtime comparison of $\text{Boolector}_{\text{dp}}$ vs. $\text{Boolector}_{\text{sc}}$ (7a), $\text{Boolector}_{\text{dp}}$ vs. $\text{Boolector}_{\text{ba}}$ (7b), and $\text{Boolector}_{\text{dp}}$ vs. $\text{Boolector}_{\text{ju}}$ (7c) on benchmark set *Selected* with 1200 seconds timeout, dual solver overhead included.

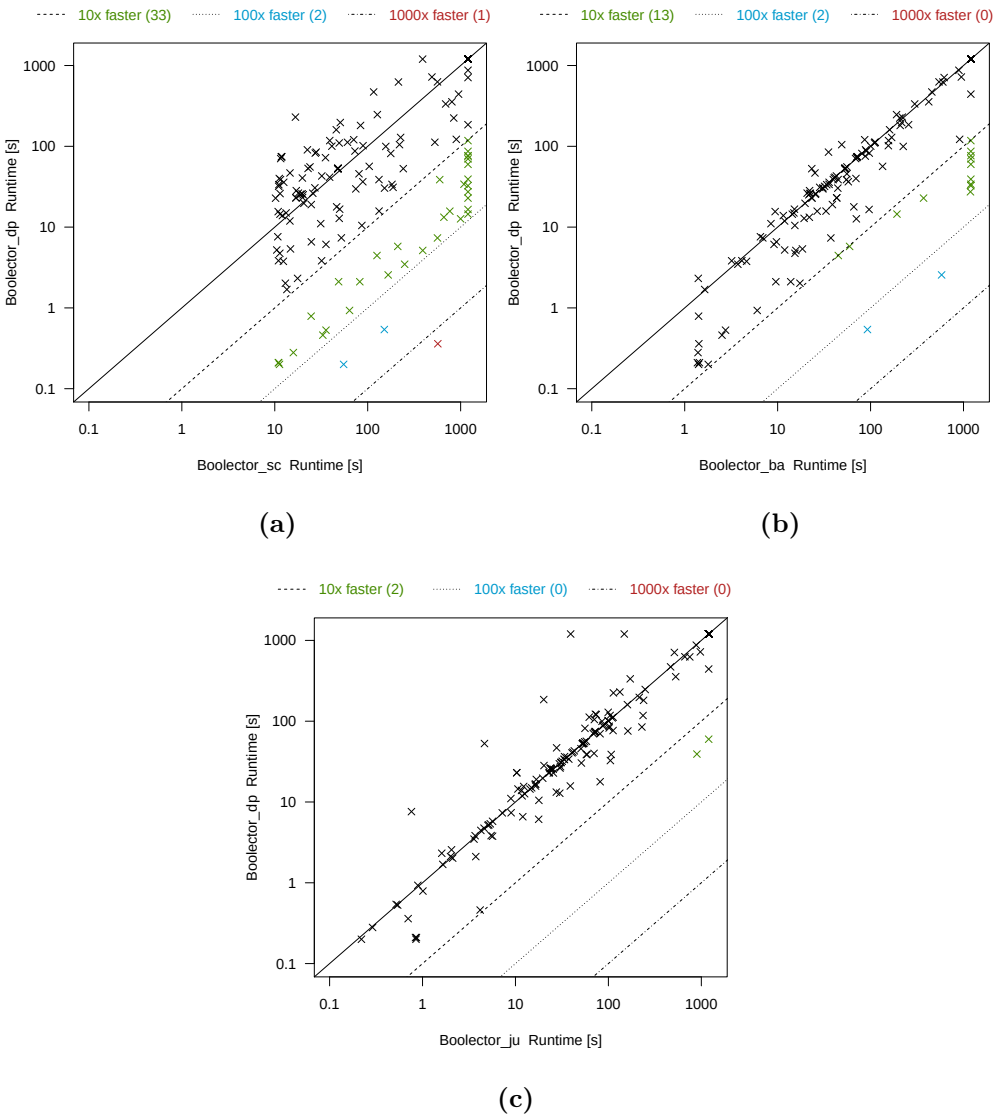


Figure 8: Runtime comparison of $\text{Boolector}_{\text{dp}}$ vs. $\text{Boolector}_{\text{sc}}$ (8a), $\text{Boolector}_{\text{dp}}$ vs. $\text{Boolector}_{\text{ba}}$ (8b), and $\text{Boolector}_{\text{dp}}$ vs. $\text{Boolector}_{\text{ju}}$ (8c) on benchmark set *Selected* with 1200 seconds timeout, dual solver overhead *not* included.

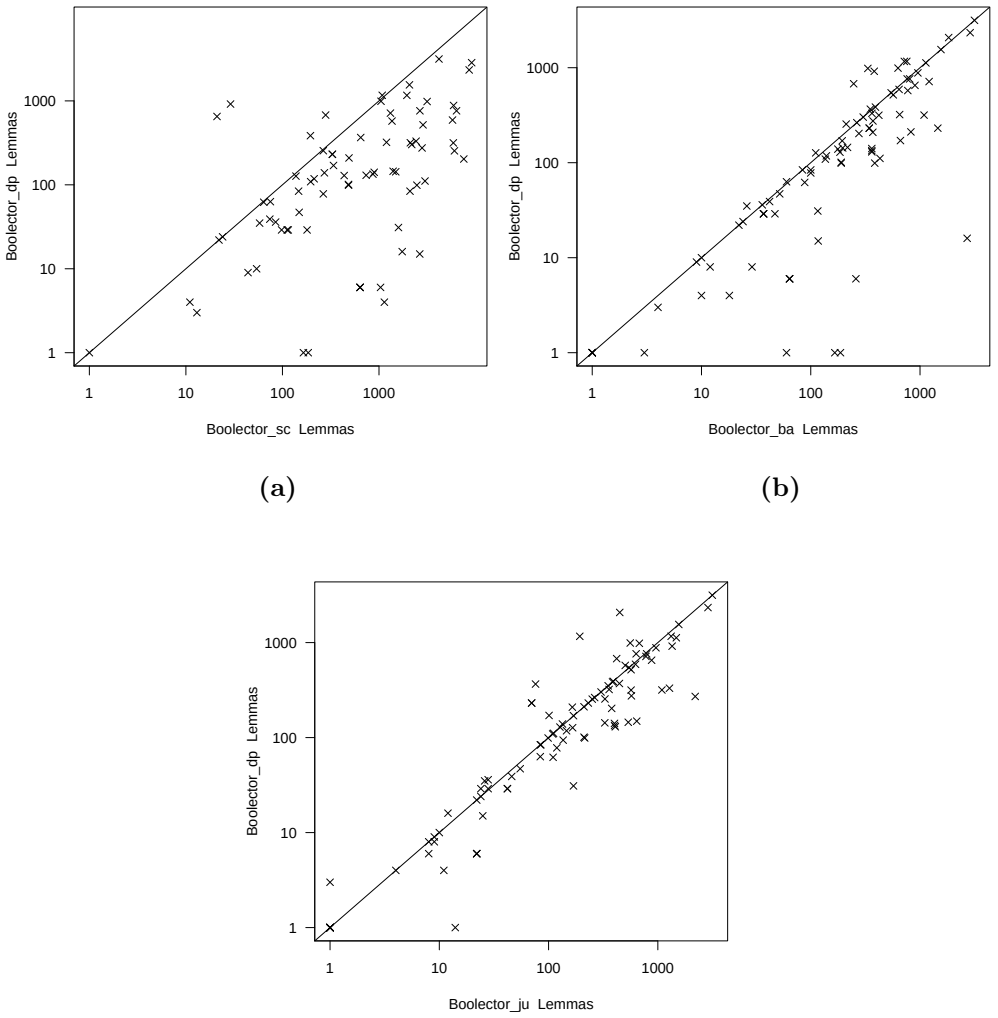


Figure 9: Comparison of the number of lemmas generated by $\text{Boolector}_{\text{dp}}$ vs. $\text{Boolector}_{\text{sc}}$ (9a), $\text{Boolector}_{\text{dp}}$ vs. $\text{Boolector}_{\text{ba}}$ (9b), and $\text{Boolector}_{\text{dp}}$ vs. $\text{Boolector}_{\text{ju}}$ (9c) on benchmark set *Selected*.

Note that the considerable difference in array model size between $\text{Boolector}_{\text{sc}}$ and $\text{Boolector}_{\text{ba}}$ is due to an optimization of procedure *search_ applies_ for_ consistency_ check* (see Section 3) introduced subsequent to [96]. In essence, given a function application $f(a)$, this optimization aims at consistency checking APPLY nodes reachable while traversing in DFS order from $f(a)$ to the primary inputs, only. In contrast, prior to that optimization it was possible that function applications irrelevant to consistency checking $f(a)$ were pulled in. The effect of this optimization is even more notable on the *Selected* benchmark set, where $\text{Boolector}_{\text{ba}}$ clearly outperforms $\text{Boolector}_{\text{sc}}$ in every aspect.

On the 113 common instances in set *Selected*, $\text{Boolector}_{\text{dp}}$ clearly outperforms $\text{Boolector}_{\text{ju}}$ and $\text{Boolector}_{\text{ba}}$ not only in terms of runtime required by the underlying SAT solver, but in the number of lemmas generated. Disregarding the dual solver overhead, $\text{Boolector}_{\text{dp}}$ shows even more improvement in terms of overall runtime than $\text{Boolector}_{\text{ju}}$. Note that without the optimization of procedure *search_ applies_ for_ consistency_ check* mentioned above, the difference in terms of overall runtime between $\text{Boolector}_{\text{ba}}$ and both optimized versions $\text{Boolector}_{\text{ju}}$ and $\text{Boolector}_{\text{dp}}$ would be even greater, i.e., comparable to the difference between both optimized versions and $\text{Boolector}_{\text{sc}}$.

4.3 Results Dual Propagation-Based Optimization

A more detailed overview of the instance-based results of our dual propagation-based approach $\text{Boolector}_{\text{dp}}$ on benchmark set *Selected* is given in Figure 7-9. Figure 7 compares the overall runtime of $\text{Boolector}_{\text{dp}}$ (incl. the overhead introduced by the dual solver) with the runtime of $\text{Boolector}_{\text{sc}}$ (7a), $\text{Boolector}_{\text{ba}}$ (7b), and $\text{Boolector}_{\text{ju}}$ (7c). Even though the dual solver overhead constitutes 31% of the total runtime of $\text{Boolector}_{\text{dp}}$, it still outperforms $\text{Boolector}_{\text{sc}}$ and $\text{Boolector}_{\text{ba}}$ on a majority of the instances and is even competitive with $\text{Boolector}_{\text{ju}}$. Disregarding the overhead of the dual solver (Figure 8), $\text{Boolector}_{\text{dp}}$ even outperforms $\text{Boolector}_{\text{ju}}$ on a majority of the instances (Figure 8c). In terms of the number of lemmas generated (Figure 9), in comparison to all three solver configurations $\text{Boolector}_{\text{sc}}$, $\text{Boolector}_{\text{ba}}$, and $\text{Boolector}_{\text{ju}}$, our dual propagation-based solver $\text{Boolector}_{\text{dp}}$ clearly shows the most notable improvement.

5 Conclusion

In this paper we introduced a dual propagation-based optimization of the lemmas on demand procedure for bit vectors with arrays as implemented in Boolector. We compared our approach with a justification-based approach similar to [2, 3]. We showed that don't care reasoning on full candidate models improves the performance of lemmas on demand considerably, Our current simulation of dual propagation is competitive with our justification-based optimization and clearly outperforms the winner of the SMT competition 2012, even though the dual solver introduces a considerable amount of overhead to the overall runtime.

Adopting a more eager dual propagation approach promises to render the dual solver overhead negligible, while further improving the overall performance by enabling partial model extraction even before a full candidate model has been generated. However, this would require an interleaved execution between the primal and the dual solver, which is rather involved to implement and subject of future work. Further, our current version of dual propagation-based partial model extraction heavily relies on incremental SAT solving under assumptions, which can benefit from dedicated data structures [83]. The integration of such SAT solver level optimization techniques is also left to future work.

Binaries of Boolector and all log files of our experimental evaluation can be found at `http://fmv.jku.at/dpjust`.

Paper D

ddSMT: A Delta Debugger for the SMT-LIB v2 Format

Published In Proceedings of the 11th International Workshop on Satisfiability Modulo Theories (SMT'13), affiliated to the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT 2013), pages 36–45, Helsinki, Finland, 2013.

Authors Aina Niemetz and Armin Biere.

Abstract Delta debugging tools automatically minimize failure-inducing input and enable efficient localization of erroneous code. In particular when debugging complex verification backends such as SMT solvers, delta debuggers provide an effective debugging approach where other debugging techniques are infeasible due to the input formula size. In this paper, we present ddSMT, a delta debugger for the SMT-LIB v2 format, which supports all SMT-LIB v2 logics and in particular handles macros and scopes defined by the commands *push* and *pop*. We introduce its architecture and describe its workflow in detail.

1 Introduction

Delta debugging algorithms [4, 32, 35, 36, 86] based on the algorithms introduced in [70, 107] typically minimize failure-inducing input by omitting parts irrelevant to the original erroneous behaviour. The resulting simplified failure-inducing input represents a minimal configuration in the sense that all of its possible subsets are necessary to cause the test to fail. Sat Modulo Theories (SMT) solvers serve as a backend for various applications in the field of e.g. deductive software verification, model checking and automated test generation. These applications heavily rely on the correctness of the underlying SMT solver – a highly complex tool, where debugging faulty behaviour becomes increasingly difficult with respect to the input formula size and structure. Rather than manually tracing error paths in order to find the actual error location, delta debugging provides means to automatically minimize input for failing SMT solvers and enables solver developers to localize failure-inducing code in a time efficient manner. Further, as shown in [32], delta debugging in combination with fuzz testing is a particularly effective approach to uncover bugs in SMT solvers.

In 2009, DeltaSMT, a delta debugger for quantifier-free logics of the previous SMT-LIB version [14] developed by our group has been presented in [32]. It is tailored to the SMT-LIB v1 language, hence incompatible with SMT-LIB v2 [16], which is a major upgrade of its predecessor. Further, DeltaSMT does not employ the original delta debugging algorithm proposed in [70], but exploits the hierarchical structure of the input formula similar to the hierarchical delta debugging approach described in [86]. Representing the input formula as a directed acyclic graph (DAG), DeltaSMT tries to simplify nodes in a breadth first search (BFS) manner. Nodes are substituted one-by-one, depending on their sort, with either constant 0, constant 1, or one of their children. Unfortunately, this substitution approach is also one of the limitations of DeltaSMT, as in the worst case, too many node-by-node substitution attempts (no matter if successful or unsuccessful) have a negative impact on the overall runtime. Further, we encountered various cases, where DeltaSMT was struggling or even unable to simplify certain input files.

More recently and independently, an update of DeltaSMT for SMT-LIB v2 by Pablo Federico Dobal and Pascal Fontaine has been released[1]. This version does not provide full SMT-LIB v2 support but syntactically extends the original tool for SMT-LIB v2 compliance, but without support for important new SMT-LIB v2 features such as quantifiers or *push* and *pop* commands. Note that in the following, we will refer to this update of DeltaSMT as DeltaSMT2.

In this paper we present ddSMT, a delta debugger for the SMT-LIB v2 format. It supports all SMT-LIB v2 logics. It is not based on DeltaSMT, but tries to overcome its limitations with a different algorithmic approach, which we will introduce in detail in the following.

[1] `http://www.verit-solver.org/veriT-toolsDownload.php`

2 The Delta Debugger ddSMT

The delta debugger ddSMT is a tool for minimizing failure-inducing input in SMT-LIB v2 format based on the exit code of a given command (typically a call to an SMT solver) when executed on that input. It is implemented in Python 3 and not only supports all SMT-LIB v2 logics, but in particular handles macros (command *define-fun*), named annotations (attribute *:named*), and scopes defined by the commands *push* and *pop*. The tool is intended to be easy to maintain and extend and further provides a dedicated, modular and standalone SMT-LIB v2 parser, which particularly should be useful for prototyping other (Python) tools working on the SMT-LIB v2 language.

2.1 Architecture

One of the challenges introduced in v2 of the SMT-LIB language is the addition of the commands *push* and *pop*, which enables scoping of assertions, and sort and function declarations. Hence, SMT-LIB v2 distinguishes between local scoping of sorted variables and variable bindings (as defined by *forall*, *exists* and *let* terms) and global scoping as defined by the commands *push* and *pop*.

Note that in the following, if distinction is needed, we refer to locally defined scopes as *term-level scopes*, and globally defined scopes as *command-level* scopes. Further note that ddSMT does not distinguish between actual functions and variables (or uninterpreted constants in first-order terminology) explicitly. Hence, in the following, if we do not make an explicit distinction, *function* may refer to either of them.

Internally the tool represents the given SMT-LIB v2 input as a tree of scopes. Each scope maintains a nesting level, a set of nested scopes, and a set of functions. Command-level scopes additionally maintain a set of commands and a set of sorts. Note that this structure enables a visibility handling of sorts and functions similar to related techniques in compiler construction, where a sorts (resp. functions) cache provides access to currently visible sorts (resp. functions) in constant time.

Example 1. To illustrate the basic internal structure of ddSMT as described above, consider the input file given in Figure 1. As shown in Figure 2, it defines two command-level scopes (the root scope at level 0 and the scope defined by given *push* and *pop* commands at level 1), and three term-level scopes defined by given *forall*, *exists* and *let* terms, respectively.

All sorts and functions defined at theory level are treated as being defined at level 0. Further, named annotations (attribute *:named*) are internally handled as if additionally a corresponding function definition had been given (in this particular case: `(define-fun z () Bool (not x))`). Commands are maintained by the scope they appear in, with a *push* command as the last command before a new scope is opened, and a *pop* command as the last command before the current

```
1  (set-logic UFNIA)
2  (declare-sort sort1 0)
3  (declare-fun x () sort1)
4  (declare-fun y () sort1)
5  (assert (= x y ))
6  (push 1)
7     (define-sort sort2 () Bool)
8     (declare-fun x () sort2)
9     (declare-fun y () sort2)
10    (assert (and (as x Bool) (as y Bool)))
11    (assert (!  (not (as x Bool)) :named z))
12    (assert z)
13 (pop 1)
14 (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
15 (check-sat)
16 (get-value ((let ((x 1) (y 1)) (= x y))))
17 (exit)
```

Figure 1: A simple example in SMT-LIB v2 format.

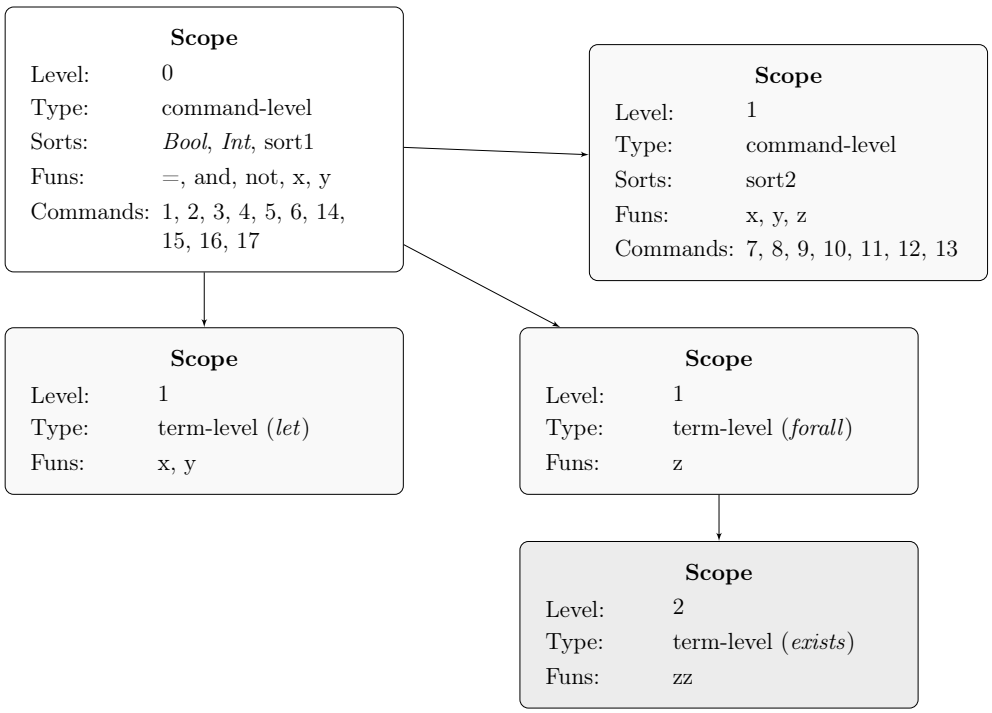


Figure 2: The basic internal structure of ddSMT given the Example in Figure 1.

scope is closed. Note that for better readability, we refer to the resp. commands in Figure 2 by the line number they appear in Figure 1.

In our example, the root scope maintains the predefined sorts *Bool* and *Int*, as well as the user-defined sort *sort1*. It further declares the predefined functions =, *and* and *not*, and the user-defined functions x and y (both of sort *sort1*). The command-level scope at level 1 maintains the user-defined sort *sort2*, and further declares functions x and y (both of sort *sort2*) and the named annotation z (of sort *Bool*). The term-level scope defined by *forall* at level 1 declares variable z of sort *Int*, whereas its nested term-level scope defined by *exists* at level 2 declares variable *zz* of sort *Int*. Finally, the term-level scope defined by *let* at level 1 declares variables x and y of sort *Int*.

2.2 General Workflow

The SMT-LIB v2 input is simplified by eliminating command-level scopes and commands, and substituting terms with simplified expressions. Note that *eliminating* scopes resp. commands refers to *substituting* nodes by *None* (the Python null object). In contrast to DeltaSMT, ddSMT does not employ a hierarchical delta debugging approach on a BFS and node-by-node substitution base, but tries to exploit the strength of the original delta debugging algorithm (a divide-and-conquer strategy) as follows. As illustrated in Figure 3, ddSMT works in rounds where each round is divided into several substitution phases. In each phase, nodes are first filtered and collected by a specific characteristic (e.g. nodes with a bit-vector sort), and then substituted using a modified version of the original delta debugging algorithm as described in Figure 8. The individual substitution phases are described as follows.

Command-Level Scope Substitution Starting with the nested scopes of the root scope, command-level scopes are eliminated level-wise, in BFS manner, until a fixpoint is reached.

Command Substitution After the command-level scope substitution phase, any command in any of the remaining command-level scopes irrelevant to the original failure-induced behaviour except the *set-logic* and *exit* commands, which are mandatory for starting and terminating SMT-LIB v2 scripts, is eliminated (while preserving the order of remaining commands) until a fixpoint is reached.

Note that in the initial round, in order to prevent lots of likely unsuccessful test runs when eliminating e.g. *declare-fun* commands previous to term substitution, ddSMT considers *assert* commands only. Further note that ddSMT does not ensure that the resulting simplified output is legal in the sense that e.g. variables must be declared previous to being used – the elimination of commands is solely tied to the exit code of the given command. This usually does not pose a problem though, as this kind of syntactically invalid input should be treated accordingly by a tool working on the SMT-LIB v2 language. In case the above

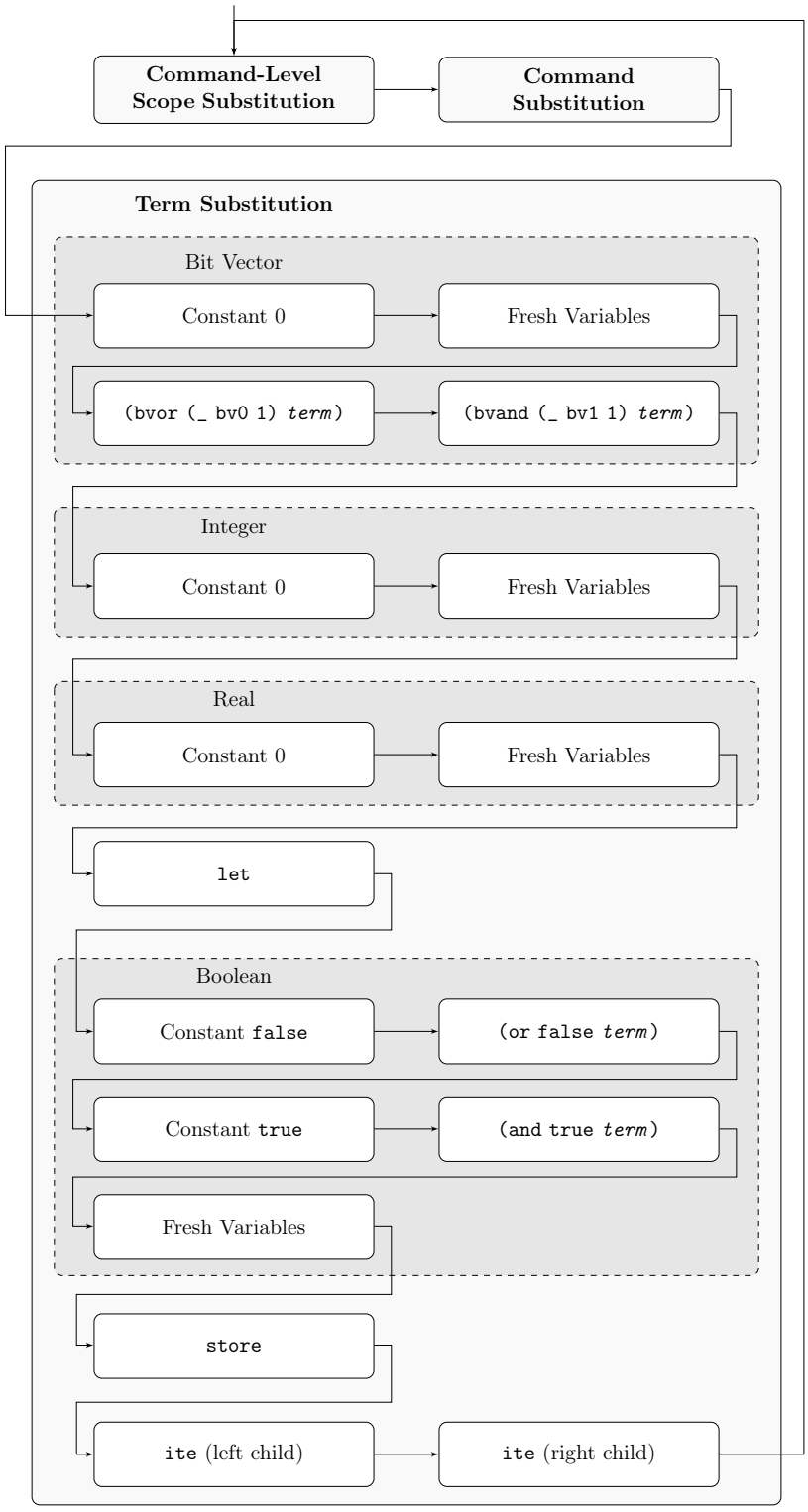


Figure 3: The general workflow and delta debugging phases of ddSMT.

behaviour poses a problem, e.g. when debugging parser related faulty behaviour, this can be easily handled by appropriate wrapper scripts (e.g. to check on specific solver output).

Term Substitution Internally, ddSMT represents SMT-LIB v2 terms as DAGs with exactly one root. The SMT-LIB v2 format defines three commands with terms as arguments: *assert*, *define-fun* and *get-value*. Commands of each of these kinds are handled separately, with *define-fun* commands being processed prior to *assert* and *get-value* commands in order to prevent redundant substitution work due to the fact that functions defined via *define-fun* are usually referenced in *assert* and *get-value* commands multiple times. For each of these sets of commands, term substitution replaces terms w.r.t. their resp. sort (and other characteristics) in several steps as indicated in Figure 3 until a fixpoint is reached. Note that individual steps (e.g. substitution of bit-vector terms with constant 0) are defined by the characteristics of both the terms to be substituted and the substitution itself. Further note that steps depending on the SMT-LIB v2 logic in use are skipped if inapplicable (e.g. the substitution of *Real* terms with constant 0 or fresh variables if given logic is not a *Reals* logic). The individual steps are described as follows.

Initially, and depending on the SMT-LIB v2 logic in use, first bit-vector, then *Int*, then *Real* terms are substituted with constant 0 and fresh variables, respectively. Additionally, if given formula is defined over the theory of *Fixed_Size_Bit_Vectors*, terms of the form `(bvor (_ bv0 1)` *`term`*`)` and `(bvand (_ bv1 1)` *`term`*`)` are replaced by their resp. child *term*. Next, *let* terms are replaced by their child term. After that, Boolean terms are substituted by constant *false*, constant *true*, and fresh variables, respectively. Subsequently, terms of the form `(or false` *`term`*`)` and `(and true` *`term`*`)` are replaced by their resp. child *term*. If the logic in use is an array logic, *store* terms are replaced by their child array terms. Finally, *ite* terms are substituted with their left and right child, respectively. Note that in those steps of the term substitution phase, where terms are replaced with a simpler expression rather than one of their child terms (e.g. substituting Boolean terms with constant *false*), constant terms are skipped. Further note that each step of the term substitution phase (e.g. substituting bit-vector terms with constant 0) is performed until a fixpoint is reached.

If any of the above substitution phases succeeded, i.e. if in any of the above phases, scopes, commands or terms have been eliminated or replaced successfully, ddSMT tries to iteratively simplify the current configuration even further until a fixpoint is reached.

Example 2. Continuing Example 1, consider the input file given in Figure 1 and an executable failing on this input by not providing support for *get-value* commands as simulated by the Shell script given in Figure 4. The input file is simplified by ddSMT in two rounds as follows.

```
1 #!/bin/sh
2 if [ ` grep -c "\<get-value\>" $1` -ne 0 ]; then exit 1; fi
3 exit 0
```

Figure 4: A simple Shell script simulating an executable failing on the input given in Figure 1.

```
1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)
5 (assert (= x y))
6 (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
7 (check-sat)
8 (get-value ((let ((x 1) (y 1)) (= x y))))
9 (exit)
```

(a) The simplified input after command-level scope substitution.

```
1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)
5 (check-sat)
6 (get-value ((let ((x 1) (y 1)) (= x y))))
7 (exit)
```

(b) The simplified input after subsequent command substitution.

Figure 5: The input of Figure 1 during the first substitution round in Example 2.

In round one, first all redundant command-level scopes are eliminated. In this case, the scope defined by the *push* and *pop* commands in line 6 and 13 is redundant. The resulting simplified input is depicted in Figure 5a.

Next, all commands irrelevant to the failure-induced behaviour are successfully eliminated. As mentioned earlier, in the first round command substitution only considers *assert* commands. Hence, commands 5 and 6 (but not command 7, which is a *check-sat* command) are eliminated. The resulting simplified input is depicted in Figure 5b.

After command substitution, ddSMT subsequently performs term substitution on argument terms of *define-fun*, *assert* and *get-value* commands, in the order specified. As the current simplified input (as depicted in Figure 5b) only contains a single *get-value* command, term substitution for *define-fun* and *assert* commands is skipped and the argument term of the *get-value* command at line 6 is the only one to be processed as follows. The original input in Figure 1 is defined over the theory of *Ints* (but not over the theory of *Fixed_Size_Bit_Vectors* or *Reals*), hence all bit-vector and *Reals* related steps are skipped. The *let* expression in line 6 contains two non-constant *Int* terms, `x` and `y`, which are first (and successfully) replaced by constant 0. The resulting simplified input is depicted in Figure 6a. As no more non-constant *Int* terms remain, subsequent substitution with fresh variables is skipped.

Next, the *let* term is successfully replaced by its child term (due to the fact that all occurrences of its variable bindings have been substituted by constant 0, previously). The resulting simplified input is depicted in Figure 6b.

Finally, the remaining non-constant Boolean term `(= 0 0)` is successfully replaced by constant *false*. As depicted in Figure 6c, in the current simplified input the only remaining term (in the argument term of the *get-value* command at line 6) is a Boolean constant. Hence, all further term substitution steps operating on Boolean and *ite* terms are skipped and the first round concludes with the simplified input depicted in Figure 6c.

In round two, the only successful substitution phase is command substitution, where commands 2, 3, and 4 are eliminated. The final result is depicted in Figure 7.

2.3 `substitute`: The Delta Debugging Core Algorithm

The core of the actual delta debugging in ddSMT is the substitution algorithm described in Figure 8. Command-level scopes and commands are substituted with *None*, whereas terms, depending on their sort, are replaced by constant 0, *false*, *true*, fresh variables, or one of their children, respectively. Each substitution phase utilizes `substitute` as follows. Given a substitution function `subst_fun` and a set of nodes filtered by some specific filter criteria (e.g. nodes with a bit-vector sort) as `superset`, this set is gradually split into `nsubsts` subsets, where the `granularity`, i.e. the number of items, of each subset initially

```
1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)
5 (check-sat)
6 (get-value ((let ((x 1) (y 1)) (= 0 0))))
7 (exit)
```

(a) The result of substituting non-constant *Int* terms with constant 0.

```
1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)
5 (check-sat)
6 (get-value ((= 0 0)))
7 (exit)
```

(b) The result of substituting the *let* term with its child term.

```
1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)
5 (check-sat)
6 (get-value (false))
7 (exit)
```

(c) The result of substituting the remaining Boolean term with constant *false*.

Figure 6: Continuing from Figure 5, all three simplified inputs are the result of individual steps of term substitution in the first round.

```
1 (set-logic UFNIA)
2 (check-sat)
3 (get-value (false))
4 (exit)
```

Figure 7: The final result of simplifying the input of Figure 1 in Example 2 after the second substitution round. In round two, commands 2, 3, and 4 are eliminated during command substitution.

```
1  def substitute (subst_fun, superset):
2      granularity = len (superset)
3      while granularity > 0:
4          nsubsets = len (superset / granularity)
5          subsets = split (superset, nsubsets)
6          for subset in subsets:
7              nsubsts = 0
8              for item in subset:
9                  if not item.is_substituted ():
10                     item.substitute_with (subst_fun (item))
11                     nsubsts + = 1
12                 if nsubsts == 0:
13                     continue
14                 dump (tmpfile)
15                 if test ():
16                     dump (outfile)
17                     subsets.delete (subset)
18                 else:     # reset substitutions of current subset
19                     restore_previous_state ()
20         superset = subsets.flatten ()
21         granularity = granularity / 2
```

Figure 8: The core substitution algorithm in ddSMT in Python-style pseudo code.

starts at `len(superset)`. Note that this basically means that in a first attempt, all nodes of `superset` will be substituted. For each `subset` of these subsets, all items are substituted by the application of the substitution function `subst_fun` to the resp. item before issuing the original command (usually a call to an SMT solver) on the current configuration. If this (so called) test run succeeds, i.e. if the exit code of the current run matches the exit code of the original configuration, the current simplified input is stored for immediate reuse in `outfile`. Otherwise, all substitutions of the current `subset` are reset and we continue with the next subset.

Note that previously substituted nodes will be skipped. This is due to the fact that `superset` initially contains either the original node (if it is yet to be substituted) or its most current substitution.

	TS	Files	Red. [%]			Time [s]		
			avg	min	max	avg	min	max
DeltaSMT	1	2	0	0	0	257	14	500
	2	95	94.0	0	99.9	49	0.1	1738
	3	5	66.6	0	93.8	12	3	34
	4	53	99.6	98.8	99.9	8	0.6	20
	5	-	-	-	-	-	-	-
	TS	**Files**	**Runs**			**Mem.** [MB]		
			avg	min	max	avg	min	max
	1	2	4051	655	7446	113	108	117
	2	95	599	5	7296	111	33	153
	3	5	608	262	1297	107	76	126
	4	53	463	4	852	128	52	142
	5	-	-	-	-	-	-	-
	TS	**Files**	**Red.** [%]			**Time** [s]		
			avg	min	max	avg	min	max
ddSMT	1	2	90.0	83.9	96.0	44	9	79
	2	95	94.7	68.2	99.9	92	0.1	1594
	3	5	80.4	66.8	87.2	23	14	35
	4	53	99.8	99.3	99.9	57	1	246
	5	5	97.4	95.7	98.3	12	5	16
	TS	**Files**	**Runs**			**Mem.** [MB]		
			avg	min	max	avg	min	max
	1	2	1412	782	2041	13	10	16
	2	95	1499	2	3790	15	10	24
	3	5	1533	1171	1764	11	10	12
	4	53	431	13	1240	28	15	42
	5	5	247	215	371	39	10	59

Table 1: Comparison between DeltaSMT (for SMT-LIB v1) and ddSMT on test sets (*TS*) 1 to 5. Test set 1 to 4 are randomly generated bit-vector formulas originally given in SMT-LIB v1, test set 5 contains non-quantifier-free test cases for CVC4. *Red.* denotes the overall reduction in percent of the original file size, *Time* denotes the overall runtime in seconds, and *Mem.* denotes the maximum resident set size in MB.

3 Experimental Evaluation

Our delta debugger ddSMT has recently been released[2] under version 3 of the General Public License (GPLv3)[3] and is currently still a work in progress. Its parser is tested on the complete SMT-LIB v2 benchmark set (available at [14]) and the delta debugger itself has been tested on a wide range of crafted instances and SMT-LIB v2 benchmarks using simple shell scripts in place of actual solver calls in order to achieve a wider distribution over the SMT-LIB v2 logics. Additionally, ddSMT has further been applied to actual failure inducing test cases encountered during the development of our solver Boolector[4], as well as the open source SMT solver CVC4[5], a joint effort between the NYU and the University of Iowa.

As of May 23rd 2013, DeltaSMT2, which we understand to be still work in progress, does not produce legal intermediate output for bit-vector logics and is thus not able to simplify any of the test cases available for Boolector. Further, DeltaSMT2 does not support non-quantifier-free logics such as AUFLIA or AUFLIRA and is hence not applicable to any of the test cases available for CVC4. Unfortunately, it was therefore not possible to evaluate the overall performance of ddSMT in comparison to DeltaSMT2. Instead, we translated quantifier-free SMT-LIB v1 input to SMT-LIB v2 and run DeltaSMT-0.2 and ddSMT-0.96-beta on various sets of test cases (*TS*) as indicated in Table 1. Test sets 1 to 4 are randomly generated bit-vector formulas originally given in SMT-LIB v1 and serve as test cases for Boolector, whereas test set 5 contains non-quantifier-free SMT-LIB v2 test cases for CVC4. Input reduction (*Red.*) is given in percent of the file size of the original input file, *Time* denotes the wall clock runtime in seconds, and *Mem.* indicates the maximum resident set size per run in MB. All experiments were performed on a 3.4 GHz Intel Core i7-2600 machine with 16GB RAM, running a 64 Bit Arch Linux OS.

Overall and even though the bit-vector test cases where originally given in SMT-LIB v1 (i.e. they do not employ SMT-LIB v2 features such as e.g. *push* and *pop* commands, which could be fully exploited by ddSMT), our first results look promising. Even for test cases, where DeltaSMT failed to simplify given input at all, ddSMT successfully achieved reductions by at least 81.1% of the original input file size. Note that except for the test cases denoted in Table 1, we currently still miss real test cases in SMT-LIB v2 logics other than QF_BV, QF_AX and QF_AUFBV. We therefore would like to encourage the SMT community to actually use ddSMT and thus further its development, and appreciate any comments, suggestions or bug reports.

[2] `http://fmv.jku.at/ddsmt`
[3] http://www.gnu.org/licenses
[4] `http://fmv.jku.at/boolector`
[5] http://cvc4.cs.nyu.edu

4 Conclusion

In this paper, we introduced our delta debugger ddSMT, a tool for minimizing failure-inducing input in SMT-LIB v2 format. It supports all SMT-LIB v2 logics and in particular handles macros, named annotations, and scopes defined by the commands *push* and *pop*. Especially in combination with fuzz testing, ddSMT provides an effective approach to find and localize bugs in tools working on the SMT-LIB v2 language.

Recently, model-based delta-debugging (and fuzzing) in the context of testing and debugging verification backends was reported to be more effective than file based delta-debugging [5], in particular in combination with option resp. configuration fuzzing. Even though the delta-debugger ddSMT presented in this paper does not work on the API level of an SMT solver directly, we believe that the "programmatic nature" of the SMT-LIB v2 format using commands allows ddSMT to be equally effective.

In future work we will compare the effectiveness of API level delta-debugging with the approach presented in this paper. We further plan to evaluate ddSMT in combination with fuzzing SMT-LIB v2 input with command-level scopes.

We want to thank Morgan Deters for providing actual test cases for the SMT solver CVC4.

Bibliography

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The key platform for verification and analysis of java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*, volume 8471 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2014.

[2] Zaher S. Andraus. *Automatic Formal Verification of Control Logic in Hardware Designs*. PhD thesis, University of Michigan, 2009.

[3] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Reveal: A formal verification tool for verilog designs. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*, pages 343–352. Springer, 2008.

[4] Cyrille Artho. Iterative delta debugging. In Hana Chockler and Alan J. Hu, editors, *Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings*, volume 5394 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2008.

[5] Cyrille Artho, Armin Biere, and Martina Seidl. Model-based testing for verification back-ends. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2013.

[6] Adrian Balint, Anton Belov, Marijn J. H. Heule, and Matti Järvisalo, editors. *SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2013.

[7] Adrian Balint, Anton Belov, Matti Järvisalo, and Carsten Sinz. Overview and analysis of the SAT challenge 2012 solver competition. *Artificial Intelligence*, 223:120–155, 2015.

[8] Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2012.

[9] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2001.

[10] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[12] Clark Barrett and Jacob Donham. Combining SAT methods with non-clausal decision heuristics. *Electr. Notes Theor. Comput. Sci.*, 125(3):3–12, 2005.

[13] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[14] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[15] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[16] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In Aarti Gupta and Darti Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[17] Sam Bayless, Celina G. Val, Thomas Ball, Holger H. Hoos, and Alan J. Hu. Efficient modular SAT solving for IC3. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 149–156. IEEE, 2013.

[18] Anton Belov, Marijn J. H. Heule, and Matti Järvisalo, editors. *SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2014.

[19] Armin Biere. AIGER Format and Toolbox. `http://fmv.jku.at/aiger`.

[20] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[21] Armin Biere. ddsexpr. `http://fmv.jku.at/ddsexpr`, 2013.

[22] Armin Biere. Yet another Local Search Solver and Lingeling and Friends Entering the SAT Competition 2014. In Adrian Balingt, Anton Belov, Marijn Heule, and Matti Järvisalo, editors, *SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, pages 39–40. University of Helsinki, 2014.

[23] Armin Biere. Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *SAT Competition 2016 – Solver and Benchmark Descriptions*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 44–45. University of Helsinki, 2016.

[24] Armin Biere and Robert Brummayer. Consistency checking of all different constraints over bit-vectors within a SAT solver. In Alessandro Cimatti and Robert B. Jones, editors, *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–4. IEEE, 2008.

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

[26] Jesse D. Bingham and Alan J. Hu. Semi-formal bounded model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 2002.

[27] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013.

[28] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[29] Robert Brummayer. *Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays*. PhD thesis, Johannes Kepler University Linz, 2009.

[30] Robert Brummayer. FuzzSMT. `http://fmv.jku.at/fuzzsmt`, 2009.

[31] Robert Brummayer and Armin Biere. Local two-level and-inverter graph minimization without blowup. In *Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06), Mikulov, Czechia*, 2006.

[32] Robert Brummayer and Armin Biere. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT'09)*, page 5. ACM, 2009.

[33] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3):165–201, 2009.

[34] Robert Brummayer, Armin Biere, and Florian Lonsing. BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking. In *Proceedings of the 1st International Workshop on Bit-Precise Reasoning, BPR 2008, affiliated with the 20th International Conference on Computer Aided Verification, CAV 2008, Princeton, NJ, USA, July 14, 2008*, 2008.

[35] Robert Brummayer and Matti Järvisalo. Testing and debugging techniques for answer set solver development. *TPLP*, 10(4-6):741–758, 2010.

[36] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010.

[37] Roberto Bruttomesso. *RTL Verification: from SAT to SMT(BV)*. PhD thesis, University of Trento, 2008.

[38] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 547–560. Springer, 2007.

[39] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153. Springer, 2010.

[40] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[41] Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental formal verification of hardware. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 135–143. FMCAD Inc., 2011.

[42] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[43] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[44] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[45] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In

Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 302–314. ACM, 2009.

[46] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[47] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[48] Leonardo de Moura and Nikolaj Bjørner. Relevancy propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.

[49] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[50] Leonardo Mendonça de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2013.

[51] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer, 2002.

[52] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 46–52. IEEE, 2013.

[53] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria,*

July 18-22, 2014. Proceedings, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[54] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134. FMCAD Inc., 2011.

[55] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[56] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245. ACM, 2002.

[57] Anders Franzen. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, 2010.

[58] Andreas Fröhlich, Armin Biere, Christoph M. Wintersteiger, and Youssef Hamadi. Stochastic local search for satisfiability modulo theories. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 1136–1143. AAAI Press, 2015.

[59] Vijay Ganesh. *Decision Procedures for Bit-Vectors, Arrays and Integers*. PhD thesis, Stanford University, 2007.

[60] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[61] Patrice Godefroid, Jonathan de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008.

[62] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.

[63] Prabhakar Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, 30(3):215–222, 1981.

[64] Alexandra Goultiaeva and Fahiem Bacchus. Exploiting QBF duality on a circuit representation. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

[65] Alexandra Goultiaeva, Martina Seidl, and Armin Biere. Bridging the gap between dual propagation and cnf-based QBF solving. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 811–814. EDA Consortium San Jose, CA, USA / ACM DL, 2013.

[66] Alberto Griggio, Quoc-Sang Phan, Roberto Sebastiani, and Silvia Tomasi. Stochastic local search for SMT: combining theory solvers with walksat. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2011.

[67] Liana Hadarean, Kshitij Bansal, Dejan Jovanovic, Clark Barrett, and Cesare Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 680–695. Springer, 2014.

[68] Trevor Alexander Hansen. *A constraint solver and its application to machine code test generation*. PhD thesis, University of Melbourne, 2012.

[69] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 58–70. ACM, 2002.

[70] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *ISSTA*, pages 135–145, 2000.

[71] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In Jim Hendler and Devika Subramanian, editors, *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA.*, pages 661–666. AAAI Press / The MIT Press, 1999.

[72] Holger H. Hoos and Thomas Stützle. Stochastic local search algorithms: An overview. In Janusz Kacprzyk and Witold Pedrycz, editors, *Springer Handbook of Computational Intelligence*, pages 1085–1105. Springer, 2015.

[73] Chung-Yang Huang and Kwang-Ting Cheng. Assertion checking by combined word-level ATPG and modular arithmetic constraint-solving techniques. In *DAC*, pages 118–123, 2000.

[74] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Efficient stochastic local search for MPE solving. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 169–174. Professional Book Center, 2005.

[75] Mahesh A. Iyer. Race: A word-level atpg-based constraints solver system for smart random simulation. In *Proceedings 2003 International Test Conference (ITC 2003), Breaking Test Interface Bottlenecks, 28 September - 3 October 2003, Charlotte, NC, USA*, pages 299–308. IEEE Computer Society, 2003.

[76] Dejan Jovanović, Clark Barrett, and Leonardo de Moura. The design and implementation of the model constructing satisfiability calculus. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 173–180. IEEE, 2013.

[77] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability.* Addison-Wesley, 2015.

[78] Nupur Kothari, Ratul Mahajan, Todd D. Millstein, Ramesh Govindan, and Madanlal Musuvathi. Finding protocol manipulation attacks. In Srinivasan Keshav, Jörg Liebeherr, John W. Byers, and Jeffrey C. Mogul, editors, *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 26–37. ACM, 2011.

[79] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory Comput. Syst.*, 59(2):323–376, 2016.

[80] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[81] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design (CAD) of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[82] Wolfgang Kunz and Dominik Stoffel. *Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[83] Jean-Marie Lagniez and Armin Biere. Factoring out assumptions to speed up MUS extraction. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 276–292. Springer, 2013.

[84] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[85] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 22–32. ACM, 2015.

[86] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 142–151. ACM, 2006.

[87] Zainalabedin Navabi. *Digital Sytem Test and Testable Design*. Springer, 2011.

[88] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *AI Magazine*, 28(3):13–30, 2007.

[89] Aina Niemetz and Armin Biere. ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories (SMT'13), affiliated to SAT'13, Helsinki, Finland*, pages 36–45, 2013.

[90] Aina Niemetz, Mathias Preiner, and Armin Biere. Turbo-charging lemmas on demand with don't care reasoning. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 179–186. IEEE, 2014.

[91] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation JSAT*, 9:53–58, 2015.

[92] Aina Niemetz, Mathias Preiner, and Armin Biere. Precise and complete propagation based local search for satisfiability modulo theories. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 199–217. Springer, 2016.

[93] Aina Niemetz, Mathias Preiner, Armin Biere, and Andreas Fröhlich. Improving local search for bit-vector logics in SMT with path propagation. In *Proceedings of the Fourth International Workshop on Design and Implementation of Formal Tools and Systems, Austin, TX, USA, September 26-27, 2015.*, pages 1–10, 2015.

[94] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *Journal of the ACM*, 53(6):937–977, 2006.

[95] Mathias Preiner. *Arrays, Lambdas and Quantifiers*. PhD thesis, Johannes Kepler University Linz, 2017.

[96] Mathias Preiner, Aina Niemetz, and Armin Biere. Lemmas on Demand for Lambdas. In Malay K. Ganai and Alper Sen, editors, *Proceedings of the Second International Workshop on Design and Implementation of Formal Tools and Systems, Portland, OR, USA, October 19, 2013.*, volume 1130 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[97] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[98] Christian Reisenberger. PBoolector: A Parallel SMT Solver for QF_BV by Combining Bit-Blasting with Look-Ahead. Master's thesis, Johannes Kepler University Linz, 2014.

[99] Roberto Sebastiani. Lazy satisability modulo theories. *JSAT*, 3(3-4):141–224, 2007.

[100] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[101] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 253–262. ACM, 2005.

[102] Gregory S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970, Volume 2*. Springer, 1983. Originally published 1970.

[103] Christoph Wintersteiger. Patch for FuzzSMT to produce SMT-LIB v2 output. `http://fmv.jku.at/fuzzsmt/fuzzsmt-smt2.patch.gz`, 2012.

[104] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research (JAIR)*, 32:565–606, 2008.

[105] Jun Yuan, Carl Pixley, and Adnan Aziz. *Constraint-based verification*. Springer, 2006.

[106] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. Deciding bit-vector formulas with mcsat. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 249–266. Springer, 2016.

[107] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.