JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:
Dipl.-Ing. Robert Daniel Brummayer Bakk. techn.

Angefertigt am:
Institut für Formale Modelle und Verifikation (FMV)

Beurteilung:
Univ.-Prof. Dr. Armin Biere (Betreuung)
Dr. Daniel Kröning

Linz, September, 2009

Abstract

The Satisfiability Modulo Theories (SMT) problem is to decide the satisfiability of a formula expressed in a (decidable) first-order background theory. In this thesis we address the problem of designing, implementing, testing, and debugging an efficient SMT solver for the quantifier-free extensional theory of arrays, combined with bit-vectors. This thesis consists of three main parts.

After an introduction into the problem and its domain, the first part considers the design of an efficient decision procedure for the quantifier-free extensional theory of arrays. We discuss the decision procedure in detail. In particular, we prove correctness, provide a complexity analysis, and discuss implementation and optimization details.

The second part focuses on the design and implementation details of our SMT solver, called Boolector. In the SMT competitions 2008 and 2009, Boolector clearly won the division of the quantifier-free theory of bit-vectors, arrays and uninterpreted functions `QF_AUFBV`. Moreover, it won the division of the quantifier-free theory of bit-vectors `QF_BV` in 2008 and achieved the second place in 2009. We discuss selected optimization techniques and features such as symbolic overflow detection, propagating unconstrained variables, and under-approximation techniques for bit-vectors.

In the last part, this thesis addresses engineering aspects such as testing and debugging techniques optimized for SMT solver development. We show that fuzzing and debugging techniques can be successfully applied to SMT solvers. We report on critical defects, such as segmentation faults and incorrect results, which we have found for state-of-the-art SMT solvers. Finally, we demonstrate the effectiveness of delta-debugging techniques in minimizing failure-inducing SMT formulas.

Zusammenfassung

Das “Satisfiability Modulo Theories (SMT) Problem” besteht darin, die Erfüllbarkeit einer Formel, welche in einer Theorie der Logik erster Ordnung ausgedrückt ist, zu entscheiden. Diese Dissertation behandelt das Problem, ein effizientes Entscheidungsprogramm für SMT Formeln der extensionalen Theorie der Arrays in Kombination mit Bit-Vektoren, ohne Quantoren, zu entwickeln. Dabei wird speziell auf den Entwurf, die Implementierung, das Testen und die Fehlerbehebung dieses Programms eingegangen. Die vorliegende Dissertation besteht aus drei Teilen.

Nach einer Einleitung, welche das behandelte Problem erläutert, widmet sich der erste Teil dieser Dissertation dem Entwurf eines Entscheidungsverfahrens für die extensionale Theorie der Arrays, ohne Quantoren. Es werden Beweise für die Korrektheit des Verfahrens, eine Komplexitäts-Analyse, sowie Implementierungs- und Optimierungsdetails behandelt.

Der zweite Teil behandelt den Entwurf und die Implementierung des SMT Entscheidungsprogramms `Boolector`. Ausgewählte Optimierungstechniken und Besonderheiten wie symbolische Überlauferkennung, Propagierung von Variablen ohne Nebenbedingungen, sowie Unter-Approximations Techniken, werden vorgestellt. Das Programm `Boolector` gewann die SMT Wettbewerbe in 2008 und 2009 in der Kategorie der extensionalen Theorie der Arrays mit Bit-Vektoren und uninterpretierten Funkionen (`QF_AUFBV`). Darüber hinaus gewann `Boolector` 2008 die Kategorie der Bit-Vektoren (`QF_BV`) und belegte 2009 den zweiten Platz.

Der dritte Teil der Dissertation widmet sich praktischen Problemen wie dem Testen und Beheben von Fehlern in SMT Entscheidungsprogrammen. Es wird gezeigt, dass randomisierte Tests und Delta-Debugging Techniken effektiv für den SMT Bereich eingesetzt werden können.

Acknowledgements

First of all, I want to thank my advisor Armin Biere. Armin was almost always available for discussions and advices which were important impulses for my research. I deeply appreciate these discussions and I will miss them. Moreover, Armin always supported me whenever I wanted to go to international conferences and workshops in order to present my current research. The scientific experiences at these conferences were important mile stones in my (still ongoing) process of becoming a researcher.

I want to thank the whole SMT community for their efforts in research. In particular, I want to thank Leonardo de Moura, Cesare Tinelli, Clark Barrett, Nikolaj Bjørner and Pete Manolios for their helpful discussions.

I want to thank my family. In particular, I want to thank my mother, Hedwig Bauböck, who was always there for me whenever I needed her. Being one of the most important persons in my life, she always supported me. Thank you so much.

I also want to thank my stepfather, Alois Starzengruber, for his support. Without his support, many things such as learning musical instruments would not have been possible in my life.

I want to thank my uncle Erhard Stuhl. Although he has never had anything to do with computers, he was always interested in my research, which was very encouraging for me.

Last but not least, I want to thank my girlfriend, Doris Knogler, who was always there for me whenever I needed her. I want to thank her for her support and for her patience. Although not a computer scientist, she was always interested in my research and even accompanied me to conferences. She always listened to me when I excitedly told her *details* of my new research ideas. Actually, Doris was so much interested that she learned the DPLL algorithm. Thank you so much, Dodo.

Robert

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Satisfiability Modulo Theories (SMT) solvers are becoming increasingly important in academia and industry. They are used to solve real world problems in domains such as formal verification [Bab08, BBC+06, HHK08, SJPS08, JES07, LQ08, SGF09], translation validation [ZPG+05, HBG04, BFG+05], scheduling [BNO+08b], and bug-finding, symbolic execution and testcase generation [BTV09, CDE08, CGP+06, TdH08, VTP09].

In particular, efficient decision procedures play an important role in formal verification. In principle, the goal of formal verification is to prove or disprove the correctness of a system with respect to a mathematical specification. Traditionally, defects in software and hardware have been found empirically by testing and simulation methods. However, the system complexity is steadily growing, which makes more sophisticated techniques like formal verification essential.

Bit-precise reasoning is a promising approach to formal verification. Traditional decision procedures use integers in order to approximate bit-vector semantics. However, such integer abstractions may not be able to detect important corner-cases such as overflows. Moreover, reasoning about non-linear operators is undecidable. Bit-precise reasoning relies on efficient decision procedures for bit-vectors. While bit-vector operations are used to represent typical word-level instructions encountered in software and hardware, bit-vector arrays can be used to represent memory components, e.g. main

memory in software and caches in hardware.

Traditionally, efficient and well-studied decision procedures such as Binary Decision Diagrams (BDDs) [McM92] and propositional satisfiability (SAT) solvers [BCCZ99] were used to perform verification on the bit-level. However, although verification problems are typically expressed on the word-level, e.g. as Register Transfer Level (RTL) design, many verification tools convert the design into a bit-level model and thus lose high-level information [KS07]. For example, on the bit-vector level, a 32 bit bit-vector division circuit can be represented as one term with two 32 bit bit-vector inputs, while on the pure bit-level it is represented as ten thousands of clauses that typically contain additional propositional variables introduced during transformation to Conjunctive Normal Form (CNF) by techniques such as the well-known Tseitin transformation [Tse83].

High-level information can be used to improve scalability and speed up verification [Sin06, Bru08, KS07]. Highly optimized decision procedures, e.g. for bit-vectors, can use word-level information to speed up solving time. For instance, a word-level decision procedure can exploit the input structure and perform term rewriting techniques that rewrite the input into an equisatisfiable but computationally less harder formula. Even solving linear bit-vector arithmetic equations, which eliminates word-level variables and parts of word-level variables, is possible [GD07]. Moreover, some instances can be fully decided on the word-level. If the decision problem cannot be fully decided on the word-level, it can then be translated to pure propositional SAT, which is an *eager* approach to SMT.

Roughly, SMT solvers can be interpreted as variants of theorem provers. However, there are two essential differences between SMT solvers and traditional theorem provers such as ACL2 [MM00], Isabelle/HOL [NPW02] and PVS [OSR92]. First, while theorem provers can also handle general interpretations and rich logic frameworks such as Higher Order Logic (HOL) that are generally undecidable, SMT solvers typically consider decidable fragments of first-order logic with specific interpretations. The price of considering undecidable theories is that theorem provers are typically interactive, i.e. they need user interactions that guide the theorem prover during the search for

the proof, while SMT solvers are fully *automatic* decision procedures, which is one of the most important features wanted in practice. Typically, system designs and specifications are iteratively refined and thus change often. This is problematic as manually or semi-automatically derived proofs have to be found by the user again, which is in contrast to fully automatic decision procedures that do not need any user interaction during the proof search. Moreover, automatic decision procedures can be integrated into theorem provers to solve sub-problems that occur during the overall proof search.

The second essential difference stems from the fact that theorem provers try to prove a theorem while SMT solvers try to satisfy a formula, which can be considered as the dual problem. On the one hand, whenever a user fails to prove a conjectured theorem with its interactive theorem prover, it is not known whether the user was not able to find the proof, or the conjectured theorem simply does not hold. On the other hand, if a formula is satisfiable, then state-of-the-art SMT solvers can provide a model. In the context of verification, a satisfiable instance corresponds to a defect that has been detected. In contrast to propositional satisfiability solvers, SMT solvers provide a concrete *word-level* counter-example that can be used in order to debug the system.

Clarke pointed out in [Cla08] that providing counter-examples is one of the main features in the success story of model checking. The ability of providing counter-examples plays an import rule in the debugging process as concrete counter-examples can be directly used to understand and correct defects. Tools such as `explain` [GKL04] have been developed to improve the quality of counter-examples [GK05] for bounded model checking of C programs [CKL04].

The theorem proving community has also recognized that the support of concrete counter-examples, i.e. finding a model [CS03, McC94, ZZ96], is important. In particular, providing a counter-example in early phases of theorem proving sessions is an important feature of theorem provers. Recently, theorem provers begin to integrate light-weight methods such as random testing [Owr06, BN04] in order to provide counter-examples for conjectured theorems up front.

1.1 SAT

The propositional satisfiability problem is the well-known classic $\mathcal{NP}$-hard problem [Coo71]. Let ϕ be a formula in Conjunctive Normal Form, i.e. ϕ consists of a conjunction of clauses, where clauses are disjunctions of literals. A literal is either a boolean variable or its negation. The SAT problem is to decide if there exists a satisfying assignment to the boolean variables of ϕ in order to satisfy all clauses.

A SAT solver is a program that takes as input a formula ϕ in CNF and decides whether it is satisfiable or not. In the satisfiable case, most state-of-the-art SAT solvers can provide a model, i.e. a satisfying assignment to the boolean variables of the formula. As each problem in $\mathcal{NP}$ can be converted to SAT in polynomial time, a SAT solver can be used as a generic decision procedure for various real world problems. Therefore, engineering efficient SAT solvers is a hot topic in research and many different SAT solving algorithms have been proposed over the last years. For a recent survey on SAT we refer the reader to [BHvMW09].

1.2 Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) problem is to decide the satisfiability of a formula expressed in a (decidable) first-order background theory that is typically a combination of first-order theories. In principle, SMT can be interpreted as an extension of propositional SAT to first-order logic. In order to remain decidable, fist-order theories are typically restricted to decidable fragments, e.g. quantifier-free fragments. Analogously to SAT, programs that decide SMT problems are called SMT solvers. SMT solvers implement highly optimized decision procedures for theories of interest. For example, important theories are bit-vectors, arrays, difference logic, linear arithmetic and uninterpreted functions. Theories are typically combined by frameworks such as Nelson-Oppen [NO79] and Shostak [Sho84].

SMT approaches can be roughly classified as *eager* or *lazy*. In the eager approach, theory constraints are eagerly compiled into the formula, i.e. an

equisatisfiable boolean formula is built up front which is passed to an efficient SAT solver afterwards. In the *lazy* SMT approach, an efficient DPLL [DLL62] SAT solver is integrated with highly optimized decision procedures for first-order theories of interest. The SAT solver is used to reason about the boolean formula structure, i.e. it enumerates assignments to the propositional skeleton of the original formula. If such an assignment is found, the theory solvers are called to decide if it is consistent modulo the background theory. If the current assignment is inconsistent, then this assignment is ruled out and the SAT solver is called again. Otherwise, a satisfying assignment has been found. In principle, decision procedures may be layered, i.e. fast (incomplete) sub-procedures are called before more expensive ones [BCF+07]. For a survey on SMT we refer the reader to [Seb07, BHvMW09, KS08, BM07].

1.3 Problem Statement and Contributions

The problem statement of this thesis is to design, implement, test, and debug an efficient SMT solver for the quantifier-free extensional theory of arrays, combined with bit-vectors. In contrast to other scientific publications about SMT, where theoretical aspects are presented solely, this thesis also provides implementation and optimization details. As already pointed out in the thesis [Eén05] of Niklas Een, implementation details are often half the result in the SAT research field. We strongly believe that is the case for research on SMT as well.

Moreover, this thesis addresses neglected engineering aspects such as testing and debugging techniques optimized for SMT solver development. We believe that implementing, testing and debugging decision procedures are non-trivial tasks which have to be considered separately. This is confirmed by our experiments in section 4.4. Although many current state-of-the art SMT solvers use decision procedures that have been extensively studied from a theoretical point of view, at the time of our tests, their concrete implementations contained defects that lead to crashes and incorrect results.

This thesis makes the following scientific contributions:

- A novel and efficient decision procedure DP_A for the quantifier-free extensional theory of arrays. The decision procedure is generic in the sense that the quantifier-free extensional theory of arrays can be combined with another decidable quantifier-free first-order theory, i.e. it is *not* limited to bit-vector arrays. We present DP_A in detail including preprocessing, formula abstraction and consistency checking. Soundness, completeness and termination proofs are used to show overall correctness. Moreover, implementation and optimizations details are presented in order to avoid practical pitfalls. Finally, experimental results confirm that our novel decision procedure DP_A is in general more efficient than previous state-of-the art decision procedures.

- Design, implementation and optimization details for Boolector which is an efficient SMT solver for bit-vectors and arrays. Boolector entered the annual SMT competition SMT-COMP'08 [BDOS08] in 2008 for the first time. It participated in the quantifier-free theory of bit-vectors `QF_BV`, and in the quantifier-free theory of bit-vectors, arrays and uninterpreted functions `QF_AUFBV`, and won both. In `QF_AUFBV` Boolector solved 16 formulas more than Z3.2, 64 more than the winner of 2007, Z3 0.1 [dMB08], and 103 more than CVC3 [BT07] version 1.5 (2007). In the SMT competition [BDOS09] 2009, Boolector won again in `QF_AUFBV` and achieved the second place in `QF_BV`. Boolector implements DP_A in order to handle bit-vector arrays.

- Novel under-approximation techniques for bit-vectors and reads on bit-vector arrays. In contrast to traditional under-approximation refinement techniques, our refinement uses the CNF layer directly, which enables a novel optimization technique called "Early Unsat Termination". We present different encoding and refinement strategies. Moreover, we show how under-approximation techniques for bit-vectors and reads on bit-vector arrays can be combined with over-approximation techniques. Experimental results confirm that our under-approximation techniques combined with over-approximation lead to a speed-up on satisfiable instances, which may, for instance, enable faster falsification.

- An adaption of known testing and debugging techniques to the SMT domain. We adapt and combine fuzz testing and delta-debugging techniques in order to efficiently test and debug SMT solvers. Our experiments show that these techniques are impressingly effective in finding and minimizing failure-inducing inputs that traditional testing techniques such as unit testing were not able to produce.

1.4 Outline

This thesis is organized as follows. In chapter two we present the novel and efficient decision procedure DP_A for the quantifier-free extensional theory of arrays. We present the decision procedure in detail, prove its correctness, provide implementation details and optimizations, and report on experimental results. In chapter three we discuss our SMT solver Boolector. In particular, we discuss its architecture and selected features such as under-approximation techniques, propagation of unconstrained variables, and symbolic overflow detection. Chapter four focuses on engineering aspects of SMT solver development such as testing and debugging. We present black-box fuzzing and delta-debugging techniques for SMT solvers, and report on experimental results. Finally, we conclude in chapter five and provide an outlook of interesting and promising topics that may be important in future research on SMT.

1.5 Previously Published Results

This thesis is partially based on some of our previous scientific publications as follows. Chapter two is based on [BB08b] and the follow-up journal article [BB09d]. In particular, we extend the journal article by additional experimental results published in [BB08b]. Chapter three is based on [BB09a]. The under-approximation techniques are published in [BB09b]. Moreover, we discuss additional features like propagation of unconstrained variables and symbolic overflow detection. Chapter four is based on [BB09c]. Furthermore,

we provide additional experiments in order to show that our techniques are not limited to bit-vector theories. Our experiments confirm that they can be successfully applied to other fragments of first-order logic such as integer difference logic and the extensional theory of arrays with uninterpreted sorts as well. Finally, Appendix A, which is partially taken from [BBL08], introduces Boolector's internal format BTOR.

Chapter 2

Extensional Theory of Arrays

2.1 Introduction

Arrays are important and widely used non-recursive data-structures, natively supported by nearly all programming languages. Software systems rely on arrays to read and store data in many different ways. In formal verification arrays are used to model main memory in software, and memory components, e.g. caches and FIFOs, in hardware systems. Typically, a flat memory model is appropriate and a memory is represented as a one-dimensional array of bit-vectors. The ability to compare arrays supports complex memory verification tasks like verifying that two algorithms leave memory in the same state. Reasoning about arrays is an important issue in formal verification and efficient decision procedures are essential.

Recently, Satisfiability Modulo Theories (SMT) solvers gained a lot of interest both in research and industry as their specific decision procedures, e.g. for the theory of arrays, turned out to be highly efficient. The SMT framework provides first-order theories to express verification conditions of interest. An SMT solver decides satisfiability of a formula expressed in a combination of first-order theories. Typically, specific decision procedures for these theories are combined by frameworks like [Bar03, NO79, Sho84]. Furthermore, theory combination does not have to be performed eagerly, but may also be delayed [BBC$^+$05, BCF$^+$06b]. If the formula is satisfiable, then

most SMT solvers can provide a model. In formal verification, a model typically represents a counter-example which may be directly used for debugging.

SMT approaches can be roughly divided into *eager* and *lazy*. In the lazy approach, the DPLL [DLL62] procedure is interleaved with highly optimized decision procedures for specific first-order theories. Decision procedures may be layered, i.e. fast (incomplete) sub-procedures are called before more expensive ones [BCF+07].

Boolean formula abstractions are enumerated by the DPLL engine. Theory solvers compute whether the enumerations produce theory conflicts or not. Boolean lemmas are used to iteratively refine the abstraction. The DPLL procedure is responsible for boolean reasoning and is the heart of this approach. Even case splits of the theory solvers may be delegated to the DPLL engine [BNOT06]. For more details about lazy SMT we refer the reader to [NOT06, Seb07].

In the eager approach, the theory constraints are eagerly compiled into the formula, i.e. an equisatisfiable boolean formula is built up front. For example, in the theory of uninterpreted functions, function symbols are eagerly replaced by fresh variables. Furthermore, congruence constraints are added to the formula. These constraints, which are also called Ackermann constraints [Ack54], are used to ensure functional consistency. Finally, the formula is checked for satisfiability once. In contrast to the lazy approach, no refinement loop is needed. For more details about SMT and first-order theories see [BHvMW09, BM07, KS08, NOT06, Seb07].

We present a novel decision procedure for the extensional theory of arrays. In our decision procedure an over-approximated formula is solved by an underlying decision procedure. If we find a spurious model, then we add a lemma to refine the abstraction. This is in the spirit of the counter-example-guided abstraction refinement approach [CGJ+03, ES03].

We consider the case where the quantifier-free extensional theory of arrays T_A is combined with a decidable quantifier-free first-order theory T_B, e.g. bit-vectors. Our decision procedure uses an abstraction refinement similar to [BDS02, dMR02, FJS03]. However, we do not use a propositional skeleton as in [Seb07], but a T_B skeleton as formula abstraction. Similar to

STP [Gan07, GD07], we replace reads by fresh abstraction variables and iteratively refine the formula abstraction. Therefore, our abstraction refinements are in the base theory T_B.

We discuss our decision procedure in a more generic context and prove soundness and completeness. As implementing a decision procedure presented at an abstract and theoretical level is nontrivial, we also provide implementation details and optimizations, in particular in combination with fixed-size bit-vectors.

The outline is as follows. In sections 2.2 and 2.3 we provide the necessary theoretical background. We introduce the extensional and non-extensional theory of arrays, and discuss preliminaries for our decision procedure. In section 2.4 we discuss preprocessing and in 2.5 formula abstraction. In section 2.6 we present a high-level overview of our decision procedure. In section 2.7 we informally introduce the main parts of our decision procedure: consistency checking and lemma generation. In section 2.8 we formally define how lemmas are generated and prove soundness. In section 2.9 we prove completeness and discuss complexity in 2.10. In section 2.11 we discuss implementation details and optimizations, and report on experimental results in section 2.12. In section 2.13 we discuss related work, and finally conclude in section 2.14.

2.2 Theory of Arrays

In principle, the theories of arrays are either extensional or non-extensional. While the extensional theories allow to reason about array elements and *arrays*, the non-extensional theories support reasoning about array elements only. McCarthy introduced the classic non-extensional theory of arrays with the help of read-over-write semantics in [McC62].

A first-order theory is typically defined by its signature and set of axioms. The theory of arrays has the signature $\Sigma_A : \{read, write, =\}$. The function $read(a, i)$ returns the value of array a at index i. Arrays are represented in a functional way. The function $write(a, i, e)$ returns array a, overwritten at index i with element e. All other elements of a remain the same. The

predicate $=$ is only applied to array *elements*. The axioms of the theory of arrays are the following:

$$\text{(A1)} \quad i = j \quad \Rightarrow \quad read(a, i) = read(a, j)$$
$$\text{(A2)} \quad i = j \quad \Rightarrow \quad read(write(a, i, e), j) = e$$
$$\text{(A3)} \quad i \neq j \quad \Rightarrow \quad read(write(a, i, e), j) = read(a, j)$$

Additionally, we assume that the theory of arrays includes the axioms of reflexivity, symmetry and transitivity of equality. If we also want to support equality on arrays, then we need an additional axiom of extensionality:

$$\text{(A4)} \quad a = b \quad \Leftrightarrow \quad \forall i \, (read(a, i) = read(b, i))$$

Note that (A1) and the implication from left to right of (A4) are instances of the function congruence axiom schema. Alternatively, (A4) can be expressed in the following way:

$$\text{(A4}') \quad a \neq b \quad \Leftrightarrow \quad \exists \lambda \, (read(a, \lambda) \neq read(b, \lambda))$$

One can interpret $(A4')$ in the way that if and only if a is unequal to b, we have to find a witness of inequality, i.e. we have to find an index λ at which the arrays differ.

We write T_A to denote the quantifier-free fragment of the *extensional* first-order theory of arrays. Moreover, we write $T_A \models \phi$ to denote that a Σ_A formula ϕ is valid in T_A (T_A-valid). A Σ_A-formula is T_A-valid if every interpretation that satisfies the axioms of T_A also satisfies ϕ.

2.3 Preliminaries

We assume that we have a decidable quantifier-free first-order theory T_B supporting equality. T_B is defined by its set of axioms and by its signature Σ_B. We assume that $\Sigma_A \cap \Sigma_B = \{=\}$, i.e. the signatures are disjoint, only equality is shared. Terms are of sort *Base*. Furthermore, we assume that we have a sound and complete decision procedure DP_B such that:

1. DP_B takes a Σ_B-formula and computes satisfiability modulo T_B.

2. If a Σ_B-formula is satisfiable, DP_B returns a B-model σ mapping terms and formulas to concrete values.

In the literature, models are satisfying assignments to variables only. However, we assume that DP_B also provides consistent assignments to arbitrary terms and formulas in the input formula. Implementing this feature is typically straightforward as we can replace variables with their concrete assignments and recursively evaluate terms and formulas to obtain each assignment. If we use a SAT solver inside DP_B in combination with some variant of Tseitin encoding [Tse83], then we can directly evaluate the assignments to the auxiliary Tseitin variables of each term resp. formula.

Our decision procedure DP_A decides satisfiability modulo $T_A \cup T_B$. The signature of T_A is augmented with the signature of T_B and the combined theory $T_A \cup T_B$ includes the axioms of T_A and T_B. Array variables and writes are terms of sort *Array*. They have sets of indices and values of sort *Base*[1] Our decision procedure takes a $(\Sigma_A \cup \Sigma_B)$-formula ϕ as input and decides satisfiability. If it is satisfiable, then our decision procedure provides an A-model. In contrast to B-models, A-models additionally provide concrete assignments to terms of sort *Array*.

2.4 Preprocessing

We apply the following two preprocessing steps to the input formula ϕ. We assume that ϕ is represented as Directed Acyclic Graph (DAG) with structural hashing enabled, i.e. syntactically identical sub-formulas and sub-terms are shared. Furthermore, we assume that inequality is represented as combination of equality and negation, i.e. $a \neq b$ is represented as $\neg(a = b)$.

1. For each equality $a = c \in \phi$ between terms of sort *Array*, we introduce a fresh variable λ of sort *Base* and two *virtual* reads $read(a, \lambda)$ and

[1] In principle, the sort of indices may differ from the sort of values. However, like most decision procedure descriptions, e.g. [SBDL01], we assume that the sorts are the same to simplify presentation and proofs. It is straightforward to generalize our decision procedure to support multiple sorts.

$read(c, \lambda)$. Then, we add the following top-level constraint:

$$a \neq c \quad \rightarrow \quad \exists \lambda.\ read(a, \lambda) \neq read(c, \lambda)$$

2. For each $write(a, i, e) \in \phi$, we add the following top-level constraint:

$$read(write(a, i, e), i) = e$$

These steps add additional top-level constraints $c_1 \ldots c_n$ to ϕ to the resulting formula π. To be more precise, π is defined as follows:

$$\pi := \phi \ \wedge \ \bigwedge_{i=1}^{n} c_i$$

The idea of preprocessing step 1 is that virtual reads are used as witness for array *inequality*. If $a \neq b$, then it must be possible to find an index λ at which the arrays contain different values. A similar usage of λ can be found in [BM07], and as k in rule *ext* [SBDL01].

The idea of preprocessing step 2 is that additional reads are introduced to enforce consistency on write values. This preprocessing step simplifies our presentation and proofs as we can focus on reads and do not have to explicitly treat write values. In contrast to preprocessing step 1, we do not expect that step 2 is performed in real implementations. In section 2.11.4 we discuss how this preprocessing step can be avoided.

Proposition 2.4.1. *ϕ and π are equisatisfiable.*

Proof. We show that each constraint c added by a preprocessing step is T_A-valid: $T_A \models c$. Therefore, conjoining these constraints to ϕ does not affect satisfiability.

Let c be an instance of preprocessing step 1. Axiom $(A4')$ asserts c. Thus, $T_A \models c$. Let c be an instance of preprocessing step 2. Axiom $(A2)$ asserts c. Thus, $T_A \models c$. □

2.4.1 If-Then-Else on Arrays

Typically, modern SMT solvers support an if-then-else on terms of sort *Array*: $cond(c, a, b)$, where c is a boolean condition. We do not explicitly treat this feature here to simplify our presentation. In principle, $cond(c, a, b)$ can always be replaced by a fresh variable d of sort *Array*, and the following constraint:

$$c \;\rightarrow\; d = a \quad \wedge \quad \neg c \;\rightarrow\; d = b$$

This preprocessing step must be performed before preprocessing step 1 and 2. In principle, our approach supports a direct integration of if-then-else on terms of sort *Array* without rewriting it up front [BB08b].

2.5 Formula Abstraction

In the following, we consider a partial formula abstraction function α. Our formula abstraction is similar to the abstraction in the lazy SMT approach, however we do *not* generate a pure propositional skeleton. We generate a T_B skeleton as formula abstraction. Similar to STP [Gan07, GD07], we replace reads by fresh variables. To be precise, our approach introduces abstraction variables of sort *Base*, and propositional abstraction variables to handle extensionality. The formula abstraction is applied after preprocessing.

Let π be the result of preprocessing a $(\Sigma_A \cup \Sigma_B)$-formula ϕ. Analogously to ϕ, we assume that π is represented as DAG with structural hashing enabled. We also assume that inequality is represented as combination of equality and negation, i.e. $a \neq b$ is represented as $\neg(a = b)$. The abstraction function α recurses down the structure of π and builds the over-approximation $\alpha(\pi)$.

For terms of sort *Array* the result of applying α is undefined. The abstraction α maps:

1. Each read $read(a, i)$ to a fresh abstraction variable of sort *Base*.

2. Each equality $a = b$ between terms of sort *Array* to a fresh propositional abstraction variable.

3. Each term $f(t_1, \ldots, t_m)$ of sort *Base* and each formula to $f(\alpha(t_1), \ldots, \alpha(t_m))$.

4. Each (non-array) variable and symbolic constant to itself.

Now, assume that we start from the boolean part of π, then terms of sort *Array* can only be reached by passing reads, and equalities between terms of sort *Array*. The abstraction function α replaces exactly these terms by fresh variables, hence the underlying terms of sort *Array* are no longer reachable in $\alpha(\pi)$.

Proposition 2.5.1. *$\alpha(\pi)$ is an over-approximating abstraction of π.*

Proof. First, we show that whenever there exists a model σ for π, then we can construct a model σ_α for $\alpha(\pi)$. We assume that σ not only returns satisfying assignments to variables, but also consistent assignments to terms and formulas in π. Given σ, σ_α can be constructed as follows: For each read r in π define $\sigma_\alpha(\alpha(r)) := \sigma(r)$. For each equality e between terms of sort *Array* in π define $\sigma_\alpha(\alpha(e)) := \sigma(e)$. For all other terms and formulas x in $\alpha(\pi)$, define $\sigma_\alpha(x) := \sigma(x)$. Obviously, σ_α satisfies $\alpha(\pi)$ as the abstraction variables are fresh and unconstrained.

Second, we show that whenever there exists a model σ_α for $\alpha(\pi)$, we can *not* always construct a model σ for π. Consider the following example. Assume that our theory T_B is the quantifier-free theory of Presburger arithmetic, i.e. *Base* represents the natural numbers. Furthermore, assume that i, λ and e are terms of sort *Base*, and a is an array with indices and values of sort *Base*. Let w be $\mathit{write}(a, i, e)$. Consider the following formula π_1:

$$w = a \wedge \mathit{read}(a, i) \neq e \wedge (w \neq a \rightarrow \mathit{read}(w, \lambda) \neq \mathit{read}(a, \lambda)) \wedge \mathit{read}(w, i) = e$$

Let q be $\alpha(w = a)$, s be $\alpha(\mathit{read}(a, i))$, t be $\alpha(\mathit{read}(w, \lambda))$, u be $\alpha(\mathit{read}(a, \lambda))$, and v be $\alpha(\mathit{read}(w, i))$. The abstraction $\alpha(\pi_1)$ results in the following formula:

$$q \wedge s \neq e \wedge (\neg q \rightarrow t \neq u) \wedge v = e$$

Obviously, we can find a model σ_α. However, π_1 is unsatisfiable as $w = a$, $\mathit{read}(a, i) \neq e$, and $\mathit{read}(w, i) = e$ can not all be assigned to $\top$. □

As $\alpha(\pi)$ is an over-approximating abstraction of π, $\alpha(\pi)$ may have additional models that are spurious in π. Therefore, we have to find a way to eliminate spurious models, which we will discuss in the next sections.

2.6 Decision Procedure

Our decision procedure DP_A uses an abstraction refinement loop similar to [BDS02, dMR02, FJS03]. However, we use a T_B skeleton as formula abstraction. Our formula abstraction introduces abstraction variables of sort *Base*. Boolean abstraction variables are only introduced to handle the extensional case.

In our decision procedure the over-approximated formula is solved by the underlying decision procedure DP_B. If we find a spurious model, then we add a lemma to refine the abstraction. In each iteration the algorithm may terminate concluding unsatisfiability, or satisfiability if the model is not spurious. However, if the model is spurious, inconsistent assignments are ruled out, and the procedure continues with the next iteration. An overview of DP_A is shown in Fig. 2.1 and Fig. 2.2.

First, we preprocess ϕ and initialize our formula refinement ξ to $\top$. In each loop iteration, we take the abstraction $\alpha(\pi)$, conjoin it with our formula refinement, and run our decision procedure DP_B. If DP_B concludes that $\alpha(\pi)$ is unsatisfiable, then we can conclude that π is unsatisfiable as α is an over-approximating abstraction. As ϕ and π are equisatisfiable, we can finally conclude that the original formula ϕ is unsatisfiable.

However, if DP_B concludes that $\alpha(\pi) \wedge \xi$ is satisfiable, it returns a satisfying assignment σ, i.e. a B-model, which can be used to build an A-model of π. As we have used an over-approximating abstraction, we have to check if this is a spurious model. We run our consistency checker on the preprocessed formula π. The consistency checker checks whether the B-model can be extended to a valid A-model or not. If the consistency checker does not find a conflict, then we can conclude that π is satisfiable. Again, as π and ϕ are equisatisfiable, we can finally conclude that ϕ is satisfiable. However, if the current assignment σ is invalid with respect to the extensional theory of

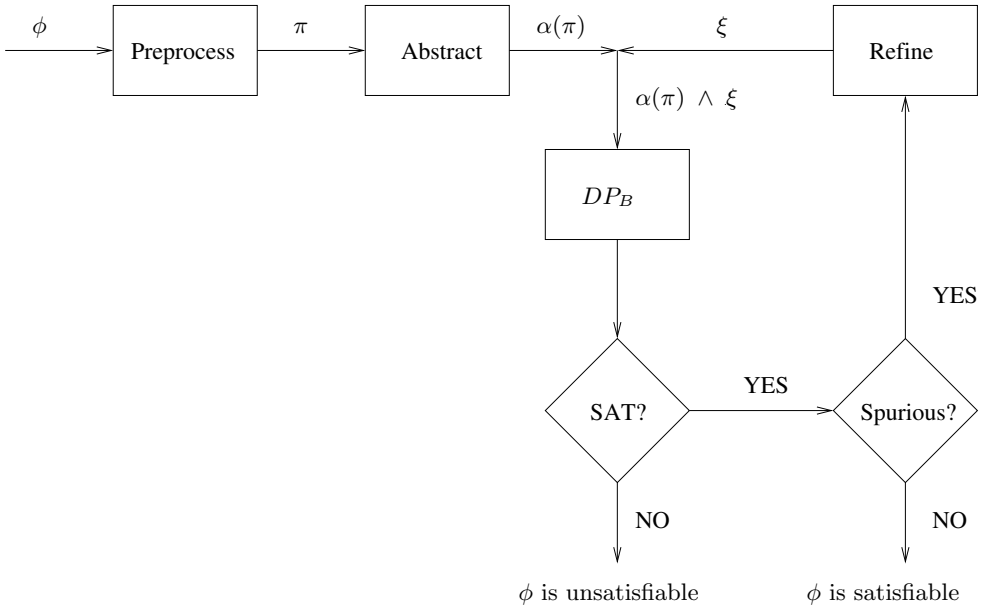

Figure 2.1: Overview of DP_A.

arrays, then we generate a lemma as formula refinement and continue with the next iteration.

2.7 Consistency Checking

In the following, we discuss the consistency checking algorithm, denoted by *consistent* in DP_A, and lemma generation, denoted by *lemma* in DP_A. A more precise description of how the lemmas are generated is part of the soundness proof in section 2.8.

The consistency checker takes π and a concrete B-model σ for $\alpha(\pi)$ and checks whether it can be extended to a valid A-model or not. In principle, the algorithm is based on read propagation and congruence checking of reads. In the first phase, the consistency checker propagates reads to other terms of sort *Array*. After the propagation has finished, the consistency checker iterates over all terms of sort *Array* in π and checks whether reads are congruent or not. If they are not, a lemma is generated to refine the abstraction.

```
procedure DP_A (φ)
    π ← preprocess (φ)
    ξ ← ⊤
    loop
        (r, σ) ← DP_B (α(π) ∧ ξ)
        if (r = unsatisfiable)
            return unsatisfiable
        if (consistent (π, σ))
            return satisfiable
        ξ ← ξ ∧ α(lemma(π, σ))
```

Figure 2.2: Algorithmic overview of DP_A.

2.7.1 Reads

First, we consider the case where reads may only be applied to array variables. Moreover, we assume that we have no writes and also no equalities between arrays. In this case no read propagation is necessary. The consistency checker iterates over all array variables and checks if congruence on reads is violated. Axiom (A1) is violated if and only if:

$$\sigma(\alpha(i)) = \sigma(\alpha(j)) \ \wedge \ \sigma(\alpha(read(a,i))) \neq \sigma(\alpha(read(a,j)))$$

In this case we generate the following lemma:

$$i = j \quad \Rightarrow \quad read(a,i) = read(a,j)$$

In principle, this is just a lazy variant of Ackermann expansion [Ack54]. In particular, Ackermann expansion is an interesting topic for SMT as it plays an important role in deciding formulas in the theory of equality and uninterpreted functions. For example, see [BCF+06a].

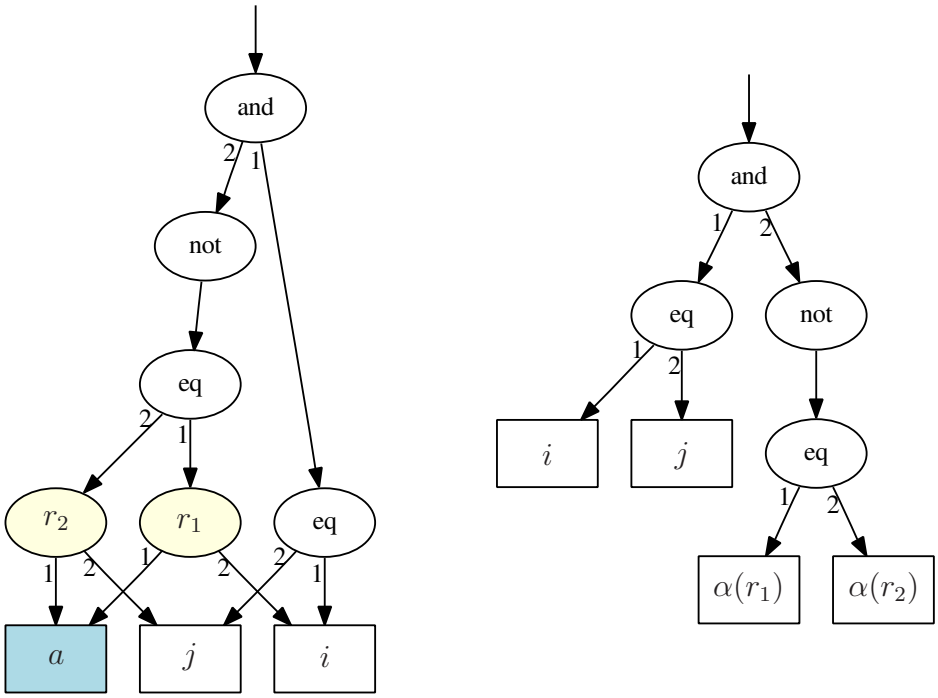

Figure 2.3: Example 1. Formula ϕ (resp. π) with array a and two reads r_1 and r_2 is shown left. The label *and* means conjunction, *not* means negation, and *eq* means equality. Edge numbers denote the ordering of the operands, e.g. r_1 is a read function where the first operand is a, and the second operand is i. The abstraction $\alpha(\pi)$ is shown right.

Example 1.

Let ϕ be as shown in Fig. 2.3. The preprocessed formula π is identical to ϕ as no preprocessing steps are necessary.

We consider the set of reads $\{r_1, r_2\}$. We run DP_B on $\alpha(\pi)$ and DP_B generates a model σ such that $\sigma(i) = \sigma(j)$ and $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$. We find an inconsistency as $\sigma(i) = \sigma(j)$, but $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$, and generate the following lemma:

$$i = j \quad \Rightarrow \quad r_1 = r_2$$

Note that in this example $\alpha(i) = i$ and $\alpha(j) = j$, but this is not the general case. Read values may also be used as read indices. Furthermore, note that our generated lemmas are in T_A. However, in order to refine our abstraction $\alpha(\pi)$, we apply the abstraction function α to each generated lemma which is not shown in our examples.

2.7.2 Writes

If we additionally consider writes in ϕ resp. π, then a term of sort *Array* is either an array variable or a write. Note that it is possible to *nest* writes. We introduce a mapping ρ which maps terms of sort *Array* to a set of reads. Note that these reads are drawn from those appearing in π. In the first phase, the consistency checker initializes ρ for each term of sort *Array*. Then, it iterates over all writes and propagates reads if necessary:

I. Map each b to its set of reads $\rho(b)$,
initialized as $\rho(b) = \{read(b, i)$ in π for some $i\}$.

D. For each $read(b, i) \in \rho(write(a, j, e))$:
if $\sigma(\alpha(i)) \neq \sigma(\alpha(j))$, add $read(b, i)$ to $\rho(a)$.

Repeat rule D until fix-point is reached, i.e. ρ does not change anymore. Fix-point computation of ρ can be implemented as post-order traversal on the array expression sub-graph starting at "top-level" writes, i.e. writes that are not overwritten by other writes. Note that read propagation in rule D is the sense of copying without removing, i.e. if we propagate a read r from source s to destination d it actually means that we copy r from $\rho(s)$ to $\rho(d)$.

In the second phase we check congruence:

> *C.* For each pair $read(b, i)$, $read(c, k)$ in $\rho(a)$:
> check *adapted* congruence axiom.

In rule C the original array congruence axiom (A1) can not be applied as $\rho(a)$ may contain propagated reads that do not read on a directly. Therefore, we have to adapt (A1). The adapted axiom is violated if and only if:

$$\sigma(\alpha(i)) = \sigma(\alpha(k)) \ \wedge \ \sigma(\alpha(read(b, i))) \neq \sigma(\alpha(read(c, k)))$$

We collect all indices $j_1^i \dots j_m^i$ that have been used as j in D while propagating $read(b, i)$. Analogously, we collect all indices $j_1^k \dots j_n^k$ that have been used as j in D while propagating $read(c, k)$. Then, we generate the following lemma:

$$i = k \ \wedge \ \bigwedge_{l=1}^{m} i \neq j_l^i \ \wedge \ \bigwedge_{l=1}^{n} k \neq j_l^k \quad \Rightarrow \quad read(b, i) = read(c, k)$$

The first big conjunction represents that $\sigma(\alpha(i))$ is different from all the assignments to the write indices on the propagation path of $read(b, i)$. Analogously, the second conjunction represents that $\sigma(\alpha(k))$ is different from all the assignments to the write indices on the propagation path of $read(c, k)$. The resulting lemma ensures that the current inconsistency can not occur anymore in future refinement iterations. Either the propagation paths change or the reads are congruent.

In this section we keep the description of our lemmas rather informal. A more precise description is part of the soundness proof in section 2.8.

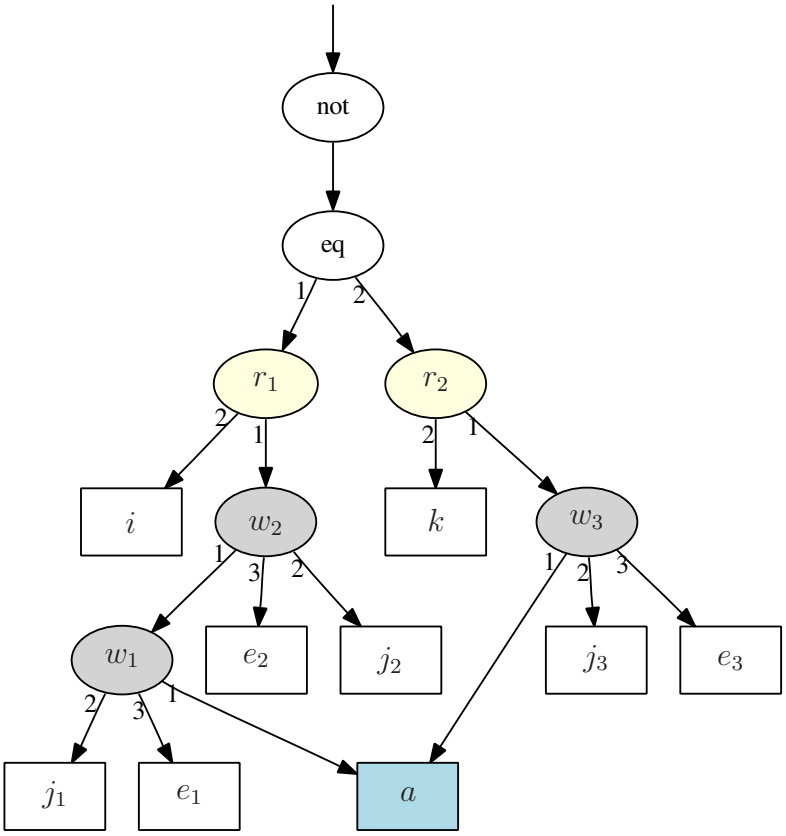

Figure 2.4: Formula ϕ for examples 2 and 3. It has one array variable a, two reads r_1 and r_2, and three writes w_1 to w_3.

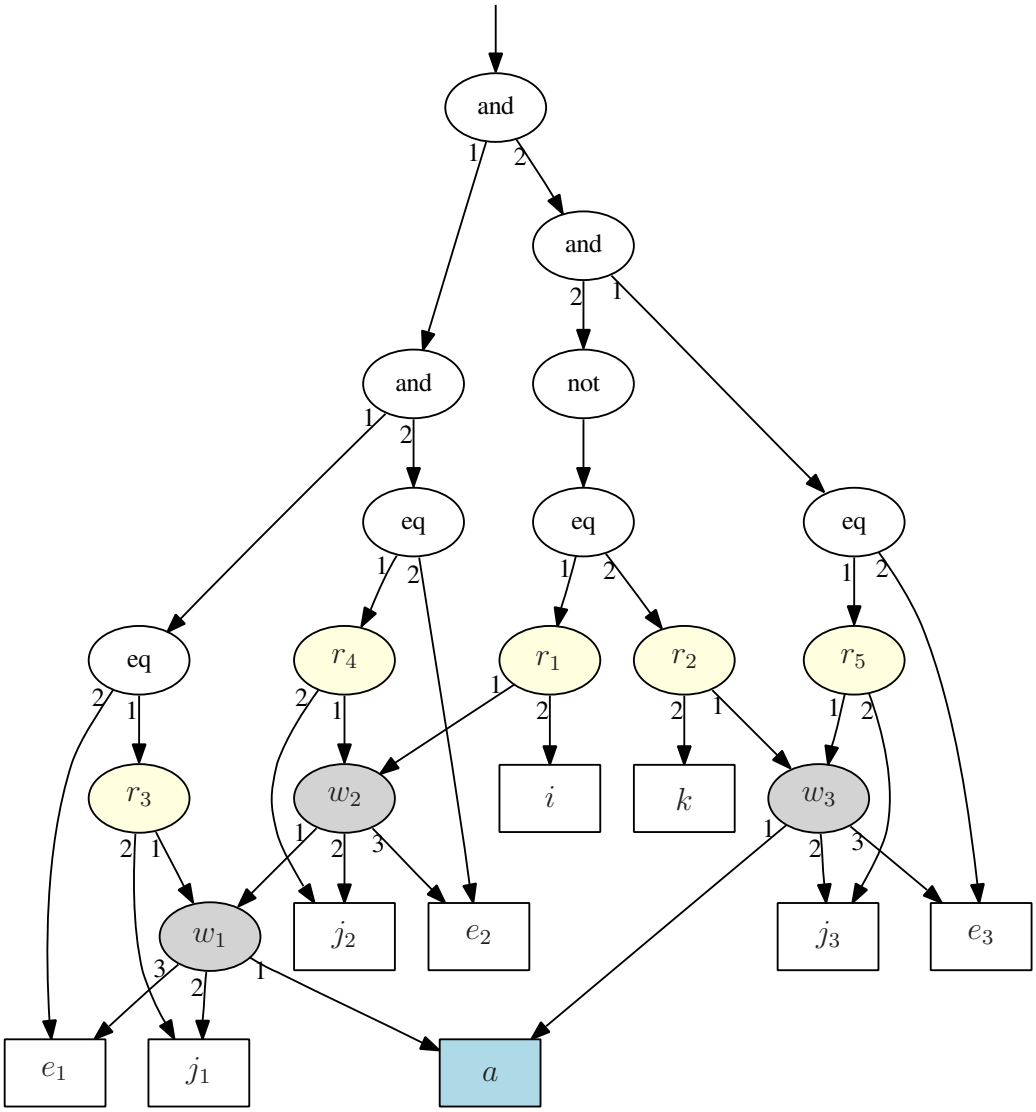

Figure 2.5: Formula π for examples 2 and 3. Reads r_3 to r_5 have been added by preprocessing step 2.

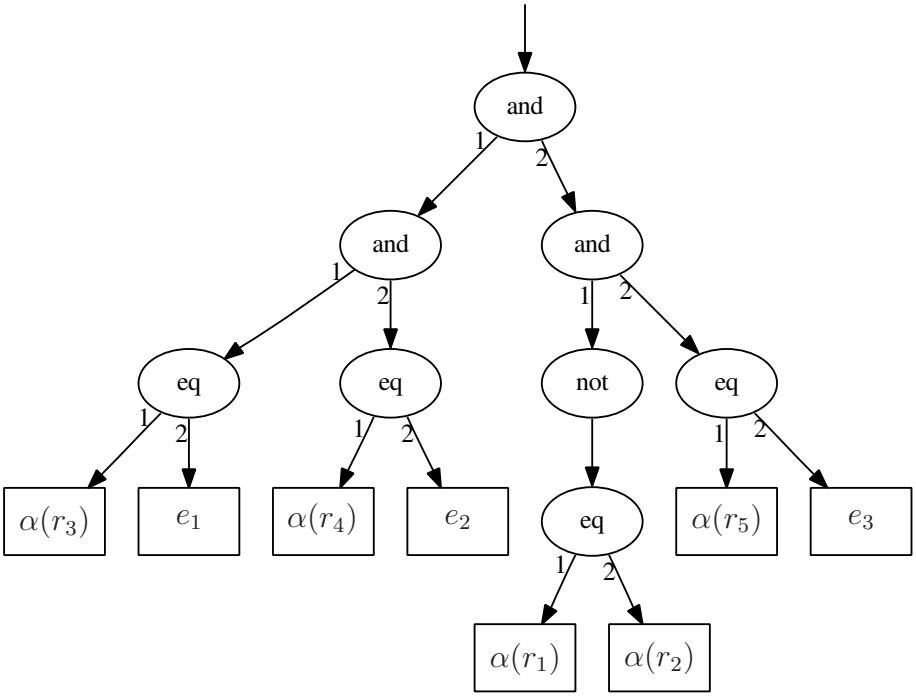

Figure 2.6: Formula $\alpha(\pi)$ for examples 2 and 3. The read indices i and k, and the write indices j_1 to j_3 are not shown as they are not reachable from the root of $\alpha(\pi)$. Initially, they are unconstrained and DP_B can assign them arbitrarily.

Example 2.

Let ϕ be as shown in Fig. 2.4. The preprocessed formula π is shown in Fig. 2.5, and $\alpha(\pi)$ is shown in Fig. 2.6.

We run DP_B on $\alpha(\pi)$ and assume that it generates a model σ such that $\sigma(i) = \sigma(k)$, $\sigma(i) \neq \sigma(j_2)$, $\sigma(i) \neq \sigma(j_1)$, $\sigma(k) \neq \sigma(j_3)$, i.e. the assignments to the write indices are different from the assignments to the read indices i and k. Furthermore, $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$. Note that if DP_B finds a model, then $\sigma(\alpha(r_3)) = \sigma(e_1)$, $\sigma(\alpha(r_4)) = \sigma(e_2)$, $\sigma(\alpha(r_5)) = \sigma(e_3)$, and $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$ has to hold.

Initially, $\rho(a) = \emptyset$, $\rho(w_1) = \{r_3\}$, $\rho(w_2) = \{r_1, r_4\}$, and $\rho(w_3) = \{r_2, r_5\}$, Read r_1 is propagated down to a as $\sigma(i) \neq \sigma(j_2)$ and $\sigma(i) \neq \sigma(j_1)$. Analogously, read r_2 is propagated down to a as $\sigma(k) \neq \sigma(j_3)$. Therefore, $\rho(a) = \{r_1, r_2\}$, $\rho(w_1) = \{r_1, r_3\}$, $\rho(w_2\} = \{r_1, r_4\}$, and $\rho(w_3) = \{r_2, r_5\}$, We check $\rho(a)$, find an inconsistency according to rule C as $\sigma(i) = \sigma(k)$, but $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$, and generate the following lemma:

$$i = k \ \wedge \ i \neq j_2 \ \wedge \ i \neq j_1 \ \wedge \ k \neq j_3 \quad \Rightarrow \quad r_1 = r_2$$

Example 3.

Again, let ϕ be as shown in Fig. 2.4. The preprocessed formula π is shown in Fig. 2.5, and $\alpha(\pi)$ is shown in Fig. 2.6.

We run DP_B on $\alpha(\pi)$ and assume that DP_B generates a model σ such that $\sigma(i) \neq \sigma(j_2)$, $\sigma(i) = \sigma(j_1)$, $\sigma(\alpha(r_1)) \neq \sigma(e_1)$, $\sigma(k) = \sigma(j_3)$, and $\sigma(\alpha(r_2)) = \sigma(\alpha(r_5))$. Again, note that $\sigma(\alpha(r_3)) = \sigma(e_1)$, $\sigma(\alpha(r_4)) = \sigma(e_2)$, $\sigma(\alpha(r_5)) = \sigma(e_3)$, and $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$.

Initially, $\rho(a) = \emptyset$, $\rho(w_1) = \{r_3\}$, $\rho(w_2) = \{r_1, r_4\}$, and $\rho(w_3) = \{r_2, r_5\}$, We propagate r_1 down to w_1 as $\sigma(i) \neq \sigma(j_2)$. Therefore, we update $\rho(w_1)$ to $\{r_1, r_3\}$. We check $\rho(w_1)$, find an inconsistency according to rule C as $\sigma(i) = \sigma(j_1)$, but $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_3))$, and generate the following lemma:

$$i = j_1 \ \wedge \ i \neq j_2 \quad \Rightarrow \quad r_1 = r_3$$

2.7.3 Equalities between Arrays

We also consider equalities between terms of sort *Array*. In particular, we add rules R and L to propagate reads over equalities between terms of sort *Array*. Furthermore, we add rule U to propagate reads upwards.

U. For each $read(b, i) \in \rho(a)$:
if $\sigma(i) \neq \sigma(j)$, add $read(b, i)$ to $\rho(write(a, j, e))$.

R. For each $a = c$, $\sigma(\alpha(a = c)) = \top$:
for each $read(b, i) \in \rho(a)$:
add $read(b, i)$ to $\rho(c)$.

L. For each $a = c$, $\sigma(\alpha(a = c)) = \top$:
for each $read(b, i) \in \rho(c)$:
add $read(b, i)$ to $\rho(a)$.

Rules R and L represent that we also have to propagate reads over equalities between terms of sort *Array* to ensure extensional consistency, but only if DP_B assigns them to $\top$. Rule U is responsible to propagate reads upwards, but only if the assignment to the write index is different from the assignment to the read index. Note, $write(a, j, e)$ must appear in π.

In order to implement our consistency checking algorithm, we need a real fix-point computation for ρ. This can be implemented by a working queue that manages future read propagations. Simple post-order traversal on the array expression sub-graph of π is no longer sufficient as reads are not only propagated downwards, but also upwards, and between equalities on terms of sort *Array*.

As soon as we consider extensionality, propagating reads upwards is necessary. Consider the following example, which is also shown in Fig. 2.11:

$$write(a, i, e_1) = write(b, j, e_2) \ \wedge \ i \neq k \ \wedge \ j \neq k \ \wedge \ read(a, k) \neq read(b, k)$$

The write indices i and j are respectively different from the read index k. Therefore, position k at array a is not overwritten with e_1. Analogously, position k at array b is not overwritten with e_2. However, in combination with $i \neq k$ and $j \neq k$, the equality between the two writes enforces that the two reads have to be extensionally congruent. Therefore, this formula is clearly unsatisfiable.

In order to detect extensionally inconsistent reads, it is necessary to propagate them to the same term of sort *Array*. Rule U enforces extensional consistency in combination with rules L and R. Reads at array positions that are not overwritten are propagated upwards to writes, and between equalities on terms of sort *Array*. In this way, extensionally inconsistent reads are propagated to the same term of sort *Array*, and rule C can detect the inconsistency. This is demonstrated in example 5. Note that upward propagation respects (A3). We propagate reads upwards only if the assignments to the read indices differ from the assignment to the write indices, i.e. the values are not overwritten.

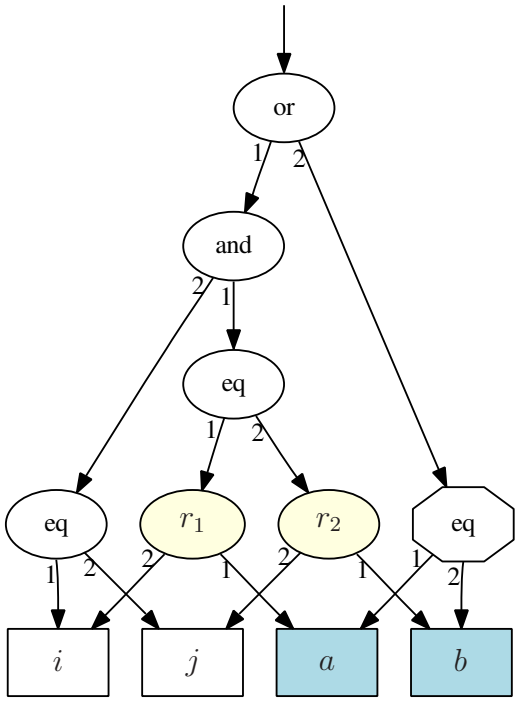

Figure 2.7: Extensional formula ϕ for example 4.

Lemmas generated upon inconsistency detection are extended as follows. Lemmas involving rule U are extended analogously to rule D. Lemmas involving rules L and R are extended in the following way. We additionally collect all array equalities $x_1^i, \ldots x_q^i$ used in rule R or L while propagating $read(b, i)$. Analogously, we additionally collect all array equalities $x_1^k, \ldots x_r^k$ used in rule R or L while propagating $read(c, k)$. The lemma is extended in the following way:

$$\ldots \wedge \bigwedge_{l=1}^{q} x_l^i \wedge \bigwedge_{l=1}^{r} x_l^k \quad \Rightarrow \quad read(b, i) = read(c, k)$$

The complete set of our rules implementing our consistency checking algorithm based on read propagation is summarized in Fig. 2.10.

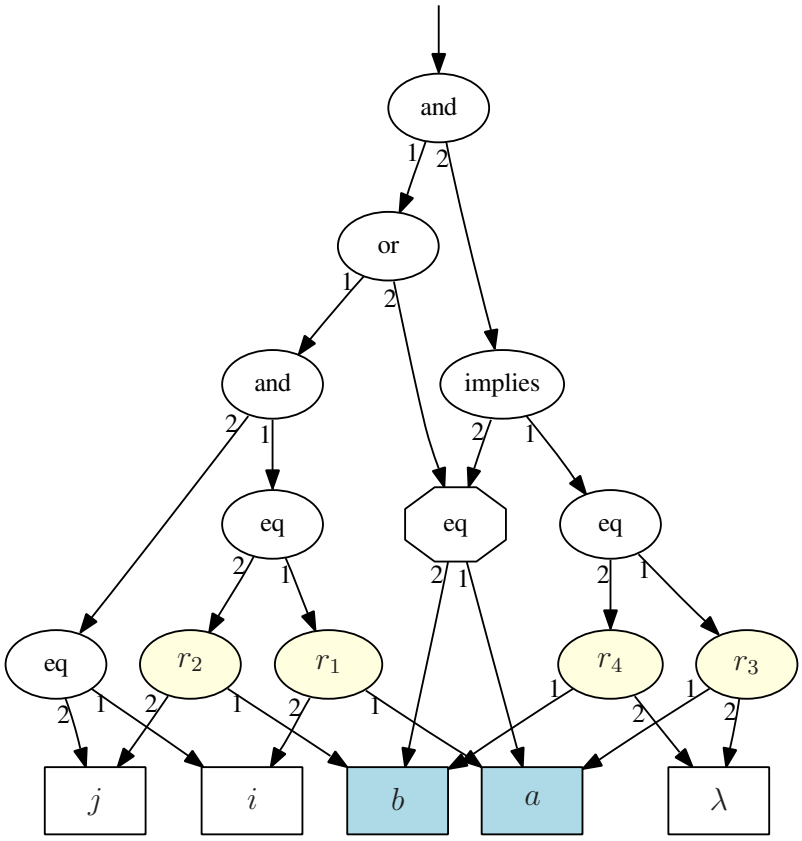

Figure 2.8: Formula π for example 4. The right part of the formula has been added by preprocessing step 1. Here we use $r_3 = r_4 \rightarrow a = b$, which is obviously equal to $a \neq b \rightarrow r_3 \neq r_4$. Note that r_3 and r_4 are virtual reads that do not occur in the original formula ϕ.

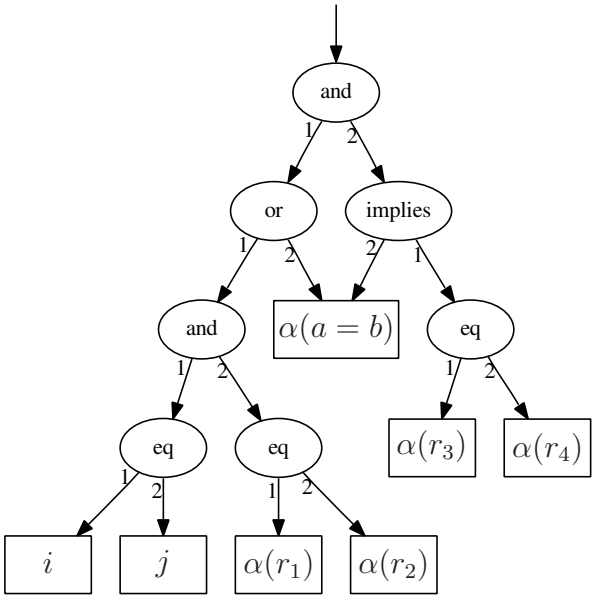

Figure 2.9: Formula $\alpha(\pi)$ for example 4.

Example 4.

Let ϕ be as in Fig. 2.7. The preprocessed formula π is shown in Fig. 2.8, and $\alpha(\pi)$ is shown in Fig. 2.9.

We run DP_B on $\alpha(\pi)$ and assume that DP_B generates a model σ such that $\sigma(i) = \sigma(j)$, $\sigma(\alpha(a = b)) = \top$, and $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$.

Initially, $\rho(a) = \{r_1, r_3\}$ and $\rho(b) = \{r_2, r_4\}$. We propagate r_1 and r_3 from a to b as $\sigma(\alpha(a = b)) = \top$ and therefore extensional consistency has to be enforced. Analogously, we propagate r_2 and r_4 from b to a. Therefore, $\rho(a) = \rho(b) = \{r_1, r_2, r_3, r_4\}$. We check $\rho(a)$, find an inconsistency according to rule C as $\sigma(i) = \sigma(j)$, but $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$. We generate the following lemma:

$$i = j \ \wedge \ a = b \quad \Rightarrow \quad r_1 = r_2$$

I. Map each b to its set of reads $\rho(b)$,
initialized as $\rho(b) = \{read(b, i) \text{ in } \pi\}$.

D. For each $read(b, i) \in \rho(write(a, j, e))$:
if $\sigma(\alpha(i)) \neq \sigma(\alpha(j))$, add $read(b, i)$ to $\rho(a)$.

U. For each $read(b, i) \in \rho(a)$:
if $\sigma(\alpha(i)) \neq \sigma(\alpha(j))$, add $read(b, i)$ to $\rho(write(a, j, e))$.

R. For each $a = c$, $\sigma(\alpha(a = c)) = \top$:
for each $read(b, i) \in \rho(a)$:
add $read(b, i)$ to $\rho(c)$.

L. For each $a = c$, $\sigma(\alpha(a = c)) = \top$:
for each $read(b, i) \in \rho(c)$:
add $read(b, i)$ to $\rho(a)$.

C. For each pair $read(b, i)$, $read(c, k)$ in $\rho(a)$:
check *adapted* congruence axiom.

Figure 2.10: Final consistency checking algorithm. Rule I initializes ρ. For each term b of sort *Array*, i.e. array variables and writes, $\rho(b)$ is initialized as set of reads that directly read on b. Rules D resp. U perform downward resp. upward propagation. Rules L resp. R perform left resp. right propagation over equalities between terms of sort *Array*. Rules D, U, R and L are repeated until fix-point is reached, i.e. ρ does not change anymore. However, in principle, consistency checking rule C may also be interleaved with D, U, R and L. This means that consistency checking is performed on-the-fly.

Example 5.

Let ϕ be as shown in Fig. 2.11. The preprocessed formula π is shown in Fig. 2.12, and $\alpha(\pi)$ is shown in Fig. 2.13.

We run DP_B on $\alpha(\pi)$ and assume that DP_B generates a model σ such that $\sigma(k) \neq \sigma(i)$, $\sigma(k) \neq \sigma(j)$, $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$, and $\sigma(\alpha(w_1 = w_2)) = \top$. Furthermore, $\sigma(\alpha(r_5)) = \sigma(e_1)$, and $\sigma(\alpha(r_6)) = \sigma(e_2)$.

We perform on-the-fly consistency checking in depth-first search manner. Initially, $\rho(a) = \{r_1\}$, $\rho(b) = \{r_2\}$, $\rho(w_1) = \{r_3, r_5\}$, and $\rho(w_2) = \{r_4, r_6\}$. We propagate r_1 up to w_1 as $\sigma(k) \neq \sigma(i)$, i.e. the value has not been overwritten by w_1. We update $\rho(w_1)$ to $\{r_1, r_3, r_5\}$. With respect to σ, we assume that the reads in $\rho(w_1)$ do not violate congruence, i.e. on-the-fly consistency checking rule C does not find a conflict in $\rho(w_1)$.

To enforce extensional consistency, we propagate r_1 from w_1 to w_2 as $\sigma(\alpha(w_1 = w_2)) = \top$. We update $\rho(w_2)$ to $\{r_1, r_4, r_6\}$. Again, we assume that the reads in $\rho(w_2)$ do not violate congruence.

Then, we propagate r_1 from w_2 down to b as $\sigma(k) \neq \sigma(j)$. We update $\rho(b)$ to $\{r_1, r_2\}$. We check $\rho(b)$, find an inconsistency according to rule C as r_1 and r_2 read on the same index, but read a different value. We generate the following lemma:

$$k \neq i \ \wedge \ k \neq j \ \wedge \ w_1 = w_2 \quad \Rightarrow \quad r_1 = r_2$$

Example 6.

Again, let ϕ be as shown in Fig. 2.11. The preprocessed formula π is shown in Fig. 2.12, and $\alpha(\pi)$ is shown in Fig. 2.13.

We run DP_B on $\alpha(\pi)$ and assume that DP_B generates a model σ such that $\sigma(i) = \sigma(j)$, $\sigma(e_1) \neq \sigma(e_2)$, and $\sigma(\alpha(w_1 = w_2)) = \top$. Furthermore, $\sigma(r_5) = \sigma(e_1)$, $\sigma(r_6) = \sigma(e_2)$, and $\sigma(\alpha(r_1)) \neq \sigma(\alpha(r_2))$.

Again, we perform on-the-fly consistency checking in depth-first search manner. Initially, $\rho(a) = \{r_1\}$, $\rho(b) = \{r_2\}$, $\rho(w_1) = \{r_3, r_5\}$, and $\rho(w_2) = \{r_4, r_6\}$. We propagate r_5 from w_1 to w_2 as $\sigma(\alpha(w_1 = w_2)) = \top$. We update $\rho(w_2)$ to $\{r_4, r_5, r_6\}$. We check $\rho(w_2)$ and find an inconsistency according to rule C as $\sigma(i) = \sigma(j)$, but $\sigma(r_5) \neq \sigma(r_6)$. We generate the following lemma:

$$i = j \ \wedge \ w_1 = w_2 \quad \Rightarrow \quad r_5 = r_6$$

2.8 Soundness

We show that our approach is sound, i.e. whenever DP_A concludes that ϕ is unsatisfiable, then ϕ is unsatisfiable modulo $T_A \cup T_B$. In particular, we show that each lemma l generated upon inconsistency detection is T_A-valid: $T_A \models l$.

First of all, we introduce a partial mapping $\chi(a, r)$, which maps a term of sort *Array* and a read $r = read(b, i)$ of sort *Base* to a propagation condition χ. If the adapted congruence axiom is violated (rule C), then the lemma is obtained by combining propagation conditions of the inconsistent reads. Thus, this section also provides a more formal definition of how lemmas are constructed in our approach.

Initially, in rule I, $\chi(a, read(a, i)) := \top$ for each read in $\rho(a)$. Otherwise, it is undefined. Whenever we propagate a read r from source s to destination

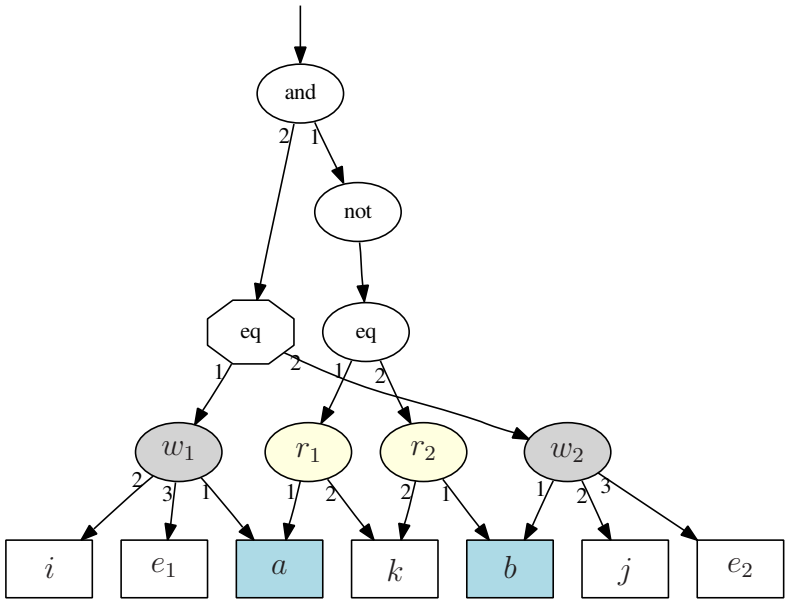

Figure 2.11: Extensional formula ϕ for examples 5 and 6.

d under the condition Δ, then we set the propagation condition $\chi(d, r)$ to $\chi(s, r) \wedge \Delta$. A propagation occurs only if $\rho(d, r)$ is undefined, i.e. the read has not been propagated to d before. For each rule in Fig. 2.10, Tab. 2.1 shows how χ is updated while propagating reads.

Lemma 2.8.1. $T_A \models \chi(d, \mathit{read}(b, i)) \Rightarrow \mathit{read}(b, i) = \mathit{read}(d, i)$

Proof. The proof is by induction over the updates to χ resp. ρ.

I**:** trivially holds: $T_A \models \top \Rightarrow \mathit{read}(b, i) = \mathit{read}(b, i)$.

Rule	s	d	Δ
I	b	b	$\top$
D	$\mathit{write}(a, j, e)$	a	$i \neq j$
U	a	$\mathit{write}(a, j, e)$	$i \neq j$
R	a	c	$a = c$
L	c	a	$a = c$

Table 2.1: Updates for χ while propagating $\mathit{read}(b, i)$ from source s to destination d under propagation condition Δ. Rule I is a special case used for initialization.

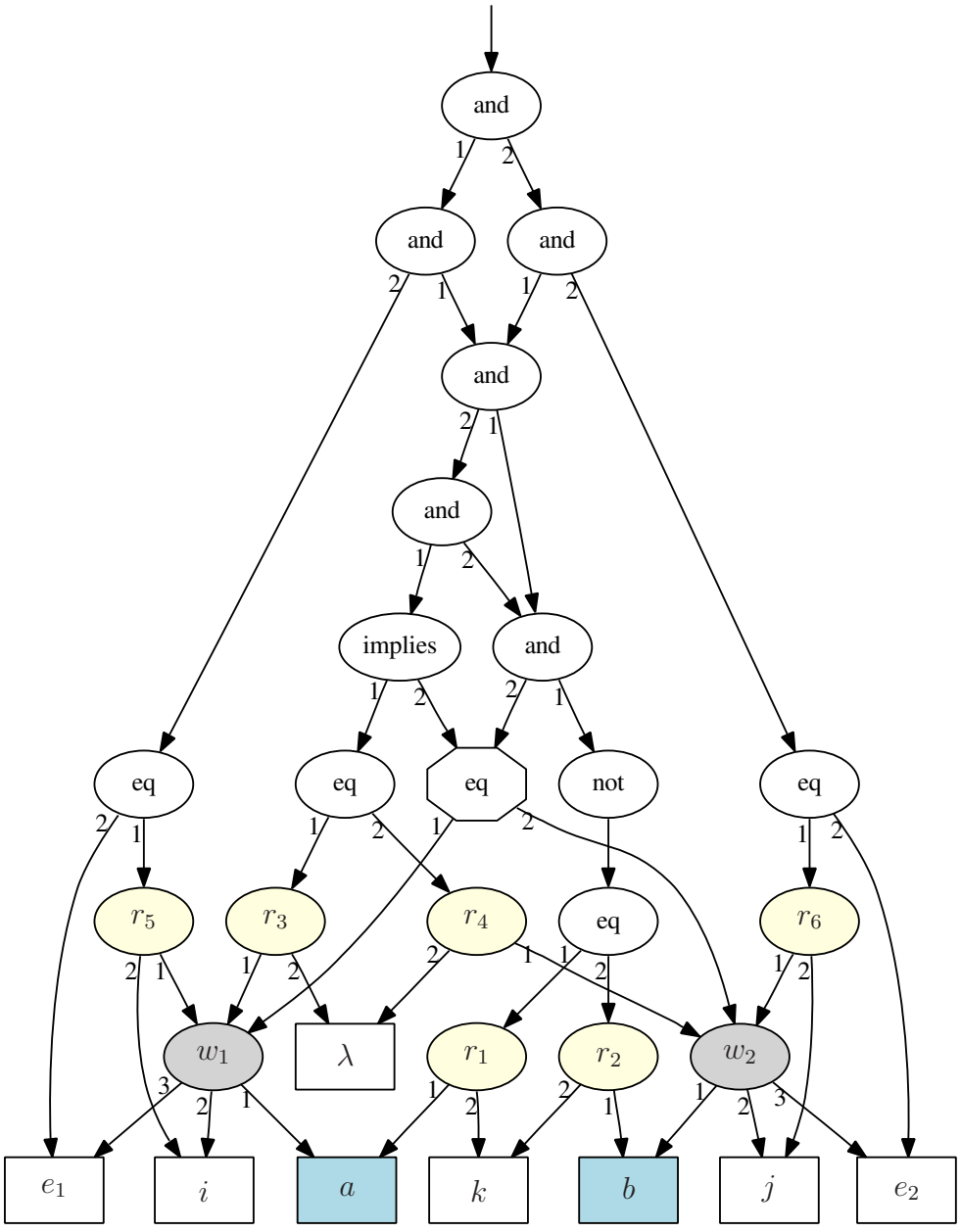

Figure 2.12: Formula π for examples 5 and 6. The two virtual reads r_3 and r_4 have been introduced by preprocessing step 1. Reads r_5 and r_6 have been introduced by preprocessing step 2 to enforce consistency on write values.

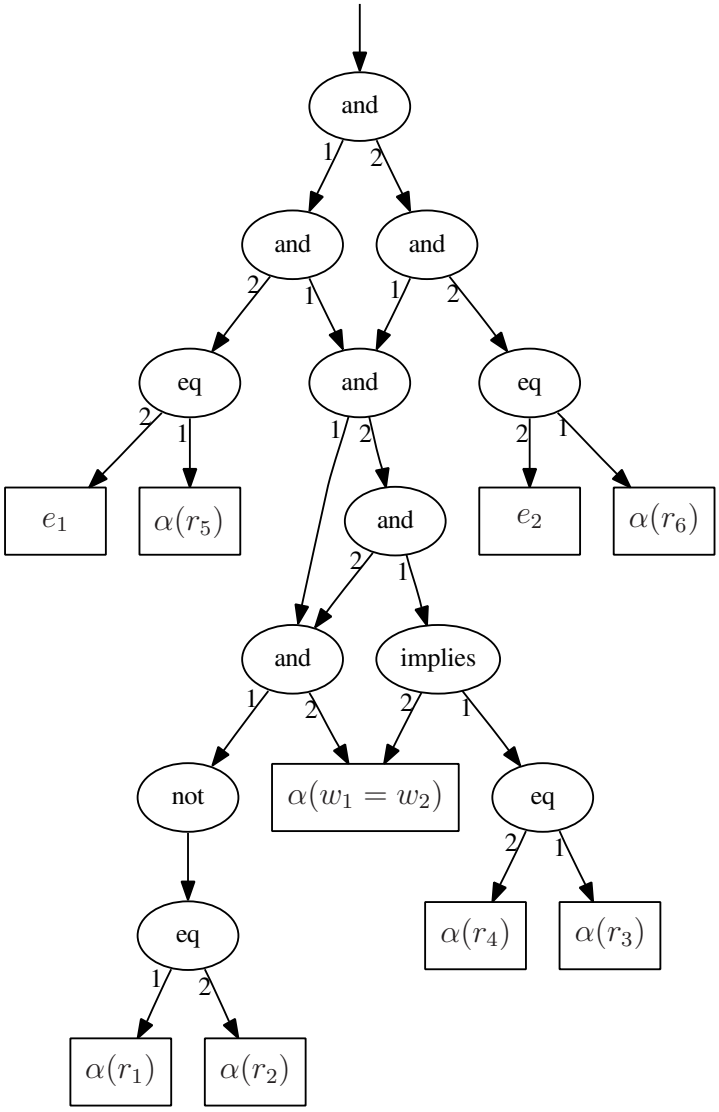

Figure 2.13: Formula $\alpha(\pi)$ for examples 5 and 6.

D: Let ψ be $\chi(write(a, j, e), read(b, i))$ and Δ be $i \neq j$. From the induction invariant we know $T_A \models \psi \Rightarrow read(write(a, j, e), i) = read(b, i)$. From (A2) we obtain $\Delta \Rightarrow read(write(a, j, e), i) = read(a, i)$. Therefore, we can conclude $T_A \models \psi \wedge \Delta \Rightarrow read(a, i) = read(write(a, j, e), i) = read(b, i)$.

U: Let ψ be $\chi(a, read(b, i))$ and Δ be $i \neq j$. From the induction invariant we know $T_A \models \psi \Rightarrow read(a, i) = read(b, i)$. From (A2) we obtain $\Delta \Rightarrow read(a, i) = read(write(a, j, e), i)$. Therefore, we can conclude $T_A \models \psi \wedge \Delta \Rightarrow read(a, i) = read(write(a, j, e), i) = read(b, i)$.

R: Let ψ be $\chi(a, read(b, i))$ and Δ be $a = c$. From the induction invariant we know $T_A \models \psi \Rightarrow read(a, i) = read(b, i)$. From (A4) we obtain that $\Delta \Rightarrow read(a, i) = read(c, i)$. Therefore, we can conclude $T_A \models \psi \wedge \Delta \Rightarrow read(a, i) = read(b, i) = read(c, i)$.

L: can be shown analogously to *R*.

□

Proposition 2.8.1. $T_A \models i = k \wedge \chi(a, read(b, i)) \wedge \chi(a, read(c, k)) \Rightarrow read(b, i) = read(c, k)$

Proof. Let ψ_b be $\chi(a, read(b, i))$ and ψ_c be $\chi(a, read(c, k))$. From lemma 2.8.1 we obtain $T_A \models \psi_b \Rightarrow read(b, i) = read(a, i)$ resp. $T_A \models \psi_c \Rightarrow read(c, k) = read(a, k)$. From (A1) we obtain that $i = k \Rightarrow read(a, i) = read(a, k)$. Therefore, $T_A \models i = k \wedge \psi_1 \wedge \psi_2 \Rightarrow read(b, i) = read(a, i) = read(a, k) = read(c, k)$ follows. □

Thus, each lemma l is T_A-valid: $T_A \models l$. Therefore, refining the formula abstraction with these lemmas does not affect satisfiability. Now, we can prove that our approach is sound:

Proposition 2.8.2. *If $DP_A(\phi) = unsatisfiable$, then ϕ is unsatisfiable modulo $T_A \cup T_B$.*

Proof. From proposition 2.4.1 we obtain that ϕ is equisatisfiable to π, and from proposition 2.5.1 we know that $\alpha(\pi)$ is an over-approximation of π. From proposition 2.8.1 we obtain that each lemma added as formula refinement does not affect satisfiability as it is T_A-valid. Thus, if DP_B concludes that $\alpha(\pi)$ is unsatisfiable, then it is sound that our overall decision procedure DP_A concludes that ϕ is unsatisfiable. □

2.9 Completeness

In order to show completeness, we define models for terms of sort *Array* and show that they respect semantics of T_A. In particular, we show that whenever DP_B finds a B-model for $\alpha(\pi)$ and the consistency checker can not find an inconsistency, the B-model in combination with ρ can be canonically extended to an A-model for π.[2] For the rest of this section we assume that DP_B found a B-model and consistency checking terminated without any violations of rule C.

First, we generalize σ for terms of sort *Base*. Let t be a term of sort *Base*:

$$\sigma(t) = \sigma(\alpha(t))$$

This means whenever we want the satisfying assignment to a term of sort *Base*, we obtain the assignment to its abstraction. Recall that for terms of sort *Base*, only reads are mapped to abstraction variables.

Now, we define models for terms of sort *Array*, which we also call σ in the following. We need exactly one arbitrary but fixed constant value of sort *Base*. In the following, we denote this value by 0. Let a be an arbitrary term of sort *Array*, and let ν be an arbitrary constant value of sort *Base*:

$$\sigma(a)(\nu) = \begin{cases} \sigma(\mathit{read}(b,i)) & \text{if exists } \mathit{read}(b,i) \in \rho(a) \text{ with } \sigma(i) = \nu \\ 0 & \text{otherwise} \end{cases}$$

This means that if we want to read the concrete value of a term a of sort

[2] We require that σ is actually defined on exactly all sub-terms of $\alpha(\pi)$. Then, σ can be extended to all sub-terms of π and ϕ. For other terms it is undefined.

Array at index ν, then we have to examine $\rho(a)$. If there is a read $read(b, i)$ where $\sigma(i) = \nu$, then $\sigma(a)(\nu)$ is $\sigma(read(b, i))$. Otherwise, the result is our constant value 0. The constant value 0 is used as default value for array elements.

First, we show that our definition of array models is well-defined. Let a be a term of sort *Array*, and let ν be a constant of sort *Base*:

Proposition 2.9.1. *Array model σ is well-defined.*

Proof. The proof is by cases. First, assume that there exists $read(b, i) \in \rho(a)$ such that $\sigma(i) = \nu$. Let $read(c, j)$ be another read $\in \rho(a)$ such that $\sigma(i) = \sigma(j) = \nu$, then $\sigma(read(b, i)) = \sigma(read(c, j))$. Otherwise, the adapted congruence would be violated. Thus, $\sigma(a)(\nu) = \sigma(read(b, i)) = \sigma(read(c, j))$, which is well-defined.

Now, assume that there does not exist a $read(b, i) \in \rho(a)$ such that $\sigma(i) = \nu$. Then, $\sigma(a)(\nu) = 0$ which is well-defined. □

Proposition 2.9.2. *Array model σ respects read semantics of T_A.*

Proof. Let a be a term of sort *Array*, and let i be a term of sort *Base*. Let $read(a, i)$ occur in π, then rule I guarantees $read(a, i) \in \rho(a)$. By definition, $\sigma(a)(\sigma(i)) = \sigma(read(a, i))$ and thus σ respects read semantics on arrays. □

Proposition 2.9.3. *Array model σ respects write semantics of T_A.*

Proof. In particular, we show that σ respects write semantics on arrays. Let a be a term of sort *Array*, and let i, j and e be terms of sort *Base*. Let $write(a, i, e)$ occur in ϕ resp. π. Let ν be a constant value of sort *Base*. Again, the proof is by cases.

First, assume that $\sigma(i) = \nu$. We know that preprocessing step 2 adds the top level constraint $read(write(a, i, e), i) = e$. Therefore, $read(write(a, i, e), i) \in \rho(write(a, i, e))$. Since σ is a B-model, $\sigma(read(write(a, i, e), i)) = \sigma(e)$. Therefore, $\sigma(write(a, i, e))(\nu) = \sigma(read(write(a, i, e), i)) = \sigma(e)$ which obviously respects update semantics of writes.

Second, assume that $\sigma(i) \neq \nu$. Assume that there exists a $read(b, j)$ with $\sigma(j) = \nu$ in $\rho(write(a, i, e))$. As $\sigma(j) = \nu$ and $\sigma(i) \neq \nu$, rule D guarantees that $read(b, j) \in \rho(a)$. Therefore, $\sigma(write(a, i, e))(\nu) = \sigma(read(b, j)) =$

$\sigma(a)(\nu)$. Analogously, in addition to $\sigma(i) \neq \nu$, assume that there exists a $read(b, j)$ with $\sigma(j) = \nu$ in $\rho(a)$. As $\sigma(j) = \nu$ and $\sigma(i) \neq \nu$, rule U guarantees that $read(b, j) \in \rho(write(a, i, e))$. Therefore, $\sigma(a)(\nu) = \sigma(read(b, j)) = \sigma(write(a, i, e))(\nu)$.

Finally, assume that there does not exist a $read(b, j)$ with $\sigma(j) = \nu$ in $\rho(write(a, i, e))$ resp. $\rho(a)$. Then, $\sigma(write(a, i, e))(\nu) = 0 = \sigma(a)(\nu)$. □

Proposition 2.9.4. *Array model σ respects extensional semantics of T_A.*

Proof. Let a and c be terms of sort *Array* with $a = c$ in ϕ. We have to show:

$$\sigma(a = c) \quad \Leftrightarrow \quad \forall \nu \, (\sigma(a)(\nu) = \sigma(c)(\nu))$$

The proof is by cases. First, we show:

$$\sigma(a \neq c) \quad \Rightarrow \quad \exists \nu \, (\sigma(a)(\nu) \neq \sigma(c)(\nu))$$

For each equality $a = c$ between terms of sort *Array*, preprocessing step 1 adds the constraint $a \neq c \Rightarrow read(a, \lambda) \neq read(c, \lambda)$. If DP_B finds a model, then there must be an assignment to λ such that $\sigma(read(a, \lambda)) \neq \sigma(read(c, \lambda))$, and therefore $\sigma(a)(\nu) \neq \sigma(c)(\nu)$ with $\nu = \sigma(\lambda)$.

Now, we show:

$$\sigma(a = c) \quad \Rightarrow \quad \forall \nu \, (\sigma(a)(\nu) = \sigma(c)(\nu))$$

Assume that there exists a $read(b, j)$ with $\sigma(j) = \nu$ in $\rho(a)$. Rule R guarantees that if $\sigma(a = c) = \top$, then $read(b, j)$ is also in $\rho(c)$. Therefore, $\sigma(a)(\nu) = \sigma(read(b, j)) = \sigma(c)(\nu)$.

Analogously, assume that there exists a $read(b, j)$ with $\sigma(j) = \nu$ in $\rho(c)$. Rule L guarantees that if $\sigma(a = c) = \top$, then $read(b, j)$ is also in $\rho(a)$. Therefore, $\sigma(c)(\nu) = \sigma(read(b, j)) = \sigma(a)(\nu)$.

Finally, assume that there does not exist a $read(b, j)$ in $\rho(a)$ resp. $\rho(c)$. Then, $\sigma(a)(\nu) = 0 = \sigma(c)(\nu)$. □

Proposition 2.9.5. *DP_A is terminating.*

Proof. In order to prove overall termination we show that only finitely many lemmas can be generated, and each loop iteration rules out at least one lemma from being regenerated. In the proof of proposition 2.10.1 we show that the number of lemmas is bounded, i.e. only finitely many lemmas can be generated.

Added lemmas can not be regenerated as for each lemma $\alpha(lemma(\pi, \sigma))$ added in the inner loop, $\sigma(\alpha(lemma(\pi, \sigma))) = \bot$. In future calls to the decision procedure DP_B, returning a satisfying assignment, this lemma has to be satisfied, and can thus not be regenerated. □

Now we are able to prove that our approach is complete.

Proposition 2.9.6. *If ϕ is unsatisfiable modulo $T_A \cup T_B$, then $DP_A(\phi) =$ unsatisfiable.*

Proof. We prove the contrapositive. From proposition 2.9.2, 2.9.3 and 2.9.4 we obtain that if DP_B finds a B-model σ for $\alpha(\pi)$ and the consistency checker does not find a conflict, then this model can always be canonically extended to an A-model of π. From proposition 2.4.1 we obtain that ϕ is equisatisfiable to π, and obviously, σ is also a model of ϕ. Therefore, in combination with proposition 2.9.5, which shows that DP_A is terminating, our approach is complete. □

2.10 Complexity

The complexity of our approach depends both on the complexity of the consistency checking algorithm and the complexity of our abstraction refinement, i.e. the upper bound on number of lemmas that have to be generated. We analyze both in the next two sub-sections.

2.10.1 Consistency Checking

The complexity of the consistency checking algorithm is quadratic in the size of π, measured in the number of rule applications that update ρ. The worst case occurs if each read is propagated to each term of sort *Array*.

In the following, we assume consistency checking is performed on-the-fly, i.e. we interleave consistency checking with read propagation. To be precise, whenever we propagate a read to its next destination d, we check if there is already a read in $\rho(d)$ that has the same assignment to its index. If this is the case, we immediately perform the consistency check.

Each rule application requires at most one update of $\rho(d)$ and one check according to rule C. One check is enough as each read in $\rho(d)$ with the same assignment to its index must have the same assignment to its read value. Otherwise, the consistency checker would have found a conflict before. Therefore, it is sufficient to compare the propagated read with only one representative read. See section 2.11.2 for a more detailed discussion.

2.10.2 Upper Bound on Number of Lemmas

Proposition 2.10.1. *The number of lemmas is bounded by* $\mathcal{O}(n^2 \cdot 2^n)$ *with* $n = |\phi|$.

Proof. Let r be the number of reads in ϕ, w be the number of writes in ϕ, and q be the number of equalities between terms of sort *Array* in ϕ. Then, π has $s = r + w + 2q$ reads, and the same number of writes and equalities between terms of sort *Array*.

By checking C on-the-fly in each update to ρ, we can make sure that propagation paths do not overlap. Therefore, writes from which the indices stem are all different. Therefore, each lemma consists of at most w comparisons of read and write indices, at most q boolean literals, exactly one read/read index comparison, and exactly one read/read value comparison. There are exactly two read indices in each lemma. Finally, each array equality contributes at most one literal. Therefore, the number of different lemmas is bounded by

$$2^q \cdot 2^w \cdot s^2 = 2^{q+w} \cdot s^2 \leq 2^n \cdot s^2 = \mathcal{O}(n^2 \cdot 2^n)$$

using $q + r + w \leq n = |\phi|$ and $s = \mathcal{O}(n)$. □

Note that rules L resp. R add abstraction variables for equalities between terms of sort *Array* to a lemma only if they have been assigned to $\top$ by DP_B.

Therefore, they can never occur negatively. Either a boolean abstraction variable occurs positively or not at all.

2.11 Implementation Details

We implemented DP_A in our SMT solver Boolector [BB09a]. Boolector is an efficient SMT solver for the quantifier-free theories of bit-vectors and arrays. Furthermore, it may also be used as model checker for safety properties in the context of bit-vectors and arrays [BBL08]. PicoSAT [Bie08] is used as backend SAT solver in DP_B.

Boolector entered the SMT competition SMT-COMP'08 [BDOS08] for the first time. It participated in the quantifier-free theory of bit-vectors `QF_BV`, and in the quantifier-free theory of bit-vectors, arrays and uninterpreted functions `QF_AUFBV`, and won both. In `QF_AUFBV` Boolector solved 16 formulas more than Z3.2, 64 more than the winner of 2007, Z3 0.1 [dMB08], and 103 more than CVC3 [BT07] version 1.5 (2007). In the SMT competition [BDOS09] 2009, Boolector won again in `QF_AUFBV` and achieved the second place in `QF_BV`.

2.11.1 Lemma Minimization

Often, lemmas constructed by our approach can be further minimized. In general, there may be many different propagation paths that lead to a theory inconsistency. Starting from the original arrays on which the reads are applied, we respectively perform a breadth-first search to the array at which the inconsistency has been detected. The breadth-first search guarantees that we respectively select one of shortest propagation paths. In general, this approach leads to stronger lemmas as the propagation conditions are directly used in the premises.

Furthermore, sometimes the premise of a lemma may be even more reduced as some parts may be already subsumed by other parts. For example, let r_1 be $read(write(a, j, e_1), i_1)$, and r_2 be $read(write(a, j, e_2), i_2)$. Consider

the following formula:

$$i_1 \neq j \ \wedge \ i_2 \neq j \ \wedge \ r_1 \neq r_2$$

Let us assume that DP_B generates an assignment such that $\sigma(i_1) \neq \sigma(j)$, $\sigma(i_2) \neq \sigma(j)$, $\sigma(i_1) = \sigma(i_2)$ and $\sigma(r_1) \neq \sigma(r_2)$. The consistency checker propagates r_1 and r_2 down to a as $\sigma(i_1) \neq \sigma(j)$ and $\sigma(i_2) \neq \sigma(j)$. It detects a conflict as $\sigma(i_1) = \sigma(i_2)$, but $\sigma(r_1) \neq \sigma(r_2)$ and generates the following lemma:

$$i_1 \neq j \ \wedge \ i_2 \neq j \ \wedge \ i_1 = i_2 \quad \Rightarrow \quad r_1 = r_2$$

This lemma can be minimized as one inequality is already subsumed by the other in combination with the equality of the read indices. For example, it can be minimized to:

$$i_1 \neq j \ \wedge \ i_1 = i_2 \quad \Rightarrow \quad r_1 = r_2$$

If we encode inequality and equality into CNF, then the shorter lemma results in fewer clauses than in the original case. This may be beneficial for the runtime of the SAT solver.

In general, the indices that are used to decide whether the read should be propagated or not, e.g. j in the previous example, can be hashed uniquely. Each lemma contains $index(read_1) = index(read_2)$ in the premise. Therefore, it is sufficient to encode that either $index(read_1)$ or $index(read_2)$ is unequal to write indices that occur in both propagation paths.

2.11.2 Implementing ρ

The set of reads ρ can be efficiently implemented by hashtables. For each term of sort *Array* a hashtable is created. A concrete assignment to a read index can be used as hash value. In the following, we assume consistency checking is performed on-the-fly, i.e. we interleave consistency checking with read propagation. Whenever we propagate a read to its next destination d, we check if there is already a read in $\rho(d)$, which has the same assignment to

its index. If this is the case, we immediately perform the consistency check.

In general, for each term a of sort *Array* it is sufficient to maintain exactly one read per concrete assignment to its read index in $\rho(a)$. If we have multiple reads in $\rho(a)$ that have the same assignment to the index values and the same assignment to the read values, then we can think of them as one equivalence class. In particular, they must have the same assignment to the read values. Otherwise, on-the-fly consistency checking would have found a conflict before. Therefore, it is sufficient to remember and propagate only one representative per class. Whenever we propagate one particular read r_1 to its next destination d and there is already a read r_2 in $\rho(d)$ which has the same assignment to index value and read value, then we do not have to insert r_1 into $\rho(d)$ as we already have the representative r_2 in $\rho(d)$. Therefore, we can skip the current propagation path of r_1 as we already have the representative r_2 in $\rho(d)$ which is also propagated further if necessary.

Implementing ρ by hashtables results in fewer propagations and consistency checks. In particular, in each propagation, consistency checking must only be performed with one representative instead of all propagated reads that have the same assignment to the index values and the same assignment to the read values.

2.11.3 Positive Array Equalities

Whenever the boolean structure of the original formula prevents DP_B from setting the boolean abstraction variable of an array equality to $\bot$, then preprocessing step 1 can be skipped. Recall that preprocessing step 1 adds one pair of virtual reads as witness for array inequality. However, if DP_B can not set the abstraction variable to $\bot$, then the virtual reads do not have to be introduced. Rules L and R are sufficient to enforce extensional consistency. The virtual reads can be safely omitted, which is beneficial for DP_A as fewer reads have to be propagated and checked. Furthermore, this decreases the number of potential lemmas. As an example, think of a formula where the boolean structure is in CNF and each array equality occurs solely positively. An example where preprocessing step 1 can be omitted is shown in Fig. 2.14.

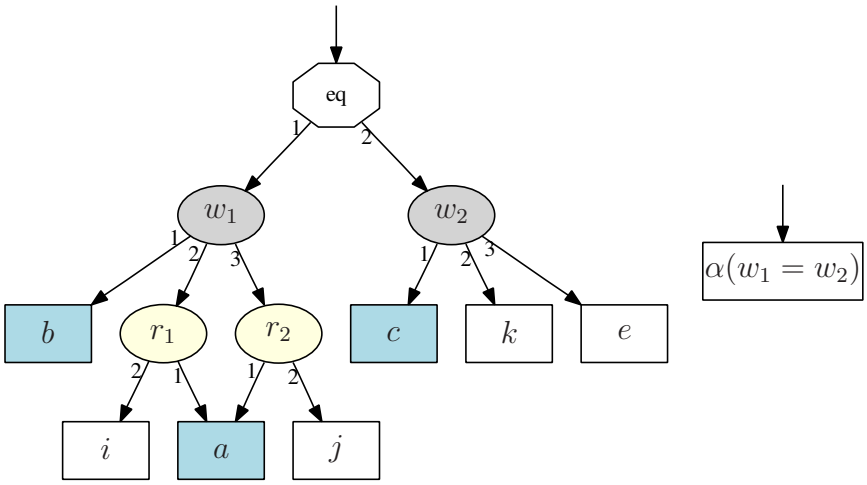

Figure 2.14: Formula ϕ resp. π for example 7 is shown left. Preprocessing step 1 is omitted as the equality between w_1 and w_2 occurs only positively. Preprocessing step 2 is omitted as it is not necessary if writes are treated and propagated as reads. The abstraction $\alpha(\pi)$ is shown right. Note that the abstraction variables $\alpha(r_1)$ and $\alpha(r_2)$ are not shown as they are not reachable from the root of the formula.

2.11.4 Treating Writes as Reads

Inside Boolector we use a polymorphic node type called *access*. A node of this type is either a read or a write. The main idea of this implementation technique is that we can treat a write as read on itself. If we have a write $write(a, i, e)$, then (A2) guarantees that a read at index i must have read value e. Instead of explicitly introducing these reads by preprocessing step 2, we can interpret a write as read on itself. We use the polymorphic functions *index* and *value* on nodes of type *access*. If we use them on a node representing a read, they return read index resp. read value. However, if we use them on a node representing $write(a, i, e)$, they return write index i resp. write value e.

Let w be $write(a, i, e)$ and let r be $read(w, i)$. If we apply *index* to r , we obtain i which is the same as applying the function directly on w. If we apply *value* to r, we get a read value which has to be equal to e. However, instead of introducing an abstraction variable for r and refining it to e, we can use e directly, which is what we obtain if we apply *value* to w directly.

In our implementation, we do not actually propagate reads. However, we propagate objects of type *access* which are either reads or writes. In this way, we also ensure congruence on write values. Therefore, we can completely skip preprocessing step 2.

Example 7.

Let ϕ resp. π be as shown left in Fig. 2.14. The abstraction $\alpha(\pi)$ is shown right.

We run DP_B on $\alpha(\pi)$ and assume that DP_B generates a model σ such that $\sigma(\alpha(w_1 = w_2)) = \top$, $\sigma(\alpha(r_1)) = \sigma(k)$, and $\sigma(\alpha(r_2)) \neq \sigma(e)$.

We perform on-the-fly consistency checking in depth-first search manner. Initially, $\rho(a) = \{r_1, r_2\}$, $\rho(w_1) = \{w_1\}$, and $\rho(w_2) = \{w_2\}$ We treat w_1 as read and propagate it to w_2 as $\sigma(\alpha(w_1 = w_2)) = \top$. We update $\rho(w_2)$ to $\{w_1, w_2\}$. We check $\rho(w_2)$, find an inconsistency according to rule C, as $\sigma(\alpha(r_1)) = \sigma(k)$, but $\sigma(\alpha(r_2)) \neq \sigma(e)$. We generate the following lemma:

$$r_1 = k \ \wedge \ w_1 = w_2 \quad \Rightarrow \quad r_2 = e$$

2.11.5 Synthesis on Demand

Boolector uses a functional representation for bit-vectors. Each term of type bit-vector is mapped to a vector of And-Inverter Graphs (AIGs) [KGP01]. For example, a term that represents the multiplication of two 32 bit-vectors is mapped to a multiplication circuit represented by AIGs. Each AIG of the resulting AIG vector represents one output of the circuit. During the construction of the AIGs, local two-level rewriting is performed [BB06]. The AIG vectors are encoded into CNF and incrementally added to the SAT solver in the backend.

In principle, each term could be encoded into AIG vectors and then encoded into CNF up front. However, this is not always necessary as a theory

conflict which leads to unsatisfiability may occur in one small part of the formula. For example, consider the following formula. Let us assume that i, j, x and y are bit-vectors with bit-width 64 and $udiv$ represents unsigned bit-vector division:

$$i \neq j \ \wedge \ read(write(a, j, udiv(x, y)), i) \neq read(a, i)$$

As $i \neq j$, the two reads have to be equal according to (A1). Hence, this formula is unsatisfiable regardless of the unsigned division. If we synthesize AIGs resp. encode to CNF up front, then the SAT solver will unnecessarily have to handle a complex 64 bit division circuit on the CNF layer, although unsatisfiability can be easily concluded without it. Therefore, we postpone AIG synthesis resp. CNF encoding of read indices, write indices, and write values. As recently as the consistency checker has to examine the concrete bit-vector assignment, the synthesis and encoding is performed, and the SAT solver is called incrementally.

In [BKO+09] for each call to the SAT solver the CNF is generated from scratch. We incrementally add clauses as required. The incremental usage of the SAT solver follows [CS03, ES03] as implemented in MiniSAT [ES04] and PicoSAT [Bie08]. Lazy synthesis is applied in our array consistency checking algorithm if we find a read or write operation, for which the bit-vector arguments have not been synthesized yet. At this point the SAT solver already determined a model for previously added clauses. This boolean model needs to be extended to new CNF variables and clauses synthesized lazily for those unsynthesized read or write indices and write values. These indices and values are unconstrained at this point and thus the SAT solver should always be able to extend the model.

However, due to restarts [GSK98] even if phase saving [Bie08, PD07] is employed, the SAT solver is free to flip a value of the original model. If this happens, the array consistency algorithm has to be aborted and restarted from scratch: the read and write propagations performed up to this point may have become invalid. In our experiments this rarely happens, simply because phase saving tries to keep the old model. However, ignoring changed

values in the old model would render the array consistency checking algorithm incomplete and unsound.

In our original version we simply restarted the array consistency checking algorithm as soon as new indices or write values have been synthesized lazily. The overhead can be avoided by adding a new API function to the SAT solver that determines if an incremental call to the SAT solver has changed the model generated in a previous SAT-call to the SAT solver. This can be implemented in a conservative way by monitoring forced assignments of variables. If a forced assignment assigns a value to a variable which is different from the previously saved assigned value to this variable, and this variable is not a new synthesis variable, then the old model has changed. In the current implementation we conservatively mark the old model as changed, even if later in the same call to the SAT solver these changes are reverted.

It would also be possible to precisely determine whether the old model has changed by saving the old model value instead of misusing saved phases. Another option is to ask the SAT solver to search for a model that extends the previous model. Both alternatives are more complex to implement, and we leave it to future work to determine whether they are more effective than our current solution.

2.11.6 CNF Encoding

In Boolector lemmas are directly added on the CNF level. No additional terms have to be created. Therefore, we need an efficient way to encode lemmas with bit-vector equalities and inequalities to SAT. For example, assume the congruence axiom (A1) is violated and we add the bit-vector lemma $i = j \;\Rightarrow\; v = w$. In order to encode this lemma to CNF, we introduce a fresh boolean variable e:

$$(i = j \;\Rightarrow\; e) \;\wedge\; (e \;\Rightarrow\; v = w)$$

This formula is equisatisfiable and can be rewritten into $(i \neq j \vee e) \wedge (\bar{e} \vee v = w)$. Let m be the number of bits of i and j, and let n be the number of bits

of v and w. We introduce m fresh variables d_k and encode $i \neq j$ as follows:

$$\bigwedge_{k=1}^{m} \left((i_k \vee j_k \vee \bar{d}_k) \ \wedge \ (\bar{i}_k \vee \bar{j}_k \vee \bar{d}_k) \right)$$

If $i_k = j_k$, then d_k is forced to $\bot$. However, if $i_k \neq j_k$, then d_k is unconstrained and can be set to $\top$. In order to encode $v = w$, we add the following clauses:

$$\bigwedge_{k=1}^{n} \left((\bar{v}_k \vee w_k \vee \bar{e}) \ \wedge \ (v_k \vee \bar{w}_k \vee \bar{e}) \right)$$

Finally, we relate the two parts through a *linking clause*:

$$e \ \vee \ \bigvee_{k=1}^{m} d_k$$

The idea of this encoding is as follows. If $i \neq j$, then they differ in at least one bit. Therefore, one d_k can be set to $\top$ to satisfy the linking clause. The variable e is now unconstrained and can be set to $\bot$. Therefore, v and w are not forced to be equal. However, if $i = j$, each d_k is $\bot$. In order to satisfy the linking clause, e has to be set to $\top$, which forces v and w to be equal.

Generally, a lemma has the following form [3], after eliminating implication:

$$\bigvee_{k=1}^{x} b_k \ \vee \ \bigvee_{k=1}^{y} (u_k \neq v_k) \ \vee \ \bigvee_{k=1}^{z} (s_k = t_k)$$

Assuming all bit-vectors have n bits, the CNF has $2 \cdot n \cdot (y + z) + 1$ clauses and $y \cdot n + z$ fresh boolean variables. The clauses are all ternary, except the linking clause of size $x + y \cdot n + z$.

In principle, bit-vector equalities and inequalities can be natively supported by SAT solvers, similar as in [BB08a]. All different constraints are directly supported by PicoSAT. This approach can simplify implementation complexity of SMT solvers that support bit-vectors and is future work.

[3] Our lemmas have exactly one inequality ($y = 1$), but we discuss the general case.

2.12 Experiments

The results of our first experimental evaluation, presented at the SMT'08 workshop [BB08b], are shown in Tab. 2.2. The following abbreviations are used for benchmark names: `sw` for `swapmem`, `wc` for `wchains`, and `dr` for `dubreva`. The last character of the benchmark name represents its status, i.e. `s` denotes satisfiable and `u` unsatisfiable.

The experiments were run on our cluster of 3 GHz Pentium IV with 2 GB main memory, running Ubuntu Linux. We set a time limit of 1800 seconds and a memory limit of 1500 MB. We compare our lemmas on demand approach to (i) an eager approach [SBDL01] implemented as rewrite system. Both approaches are implemented in Boolector 0.0. Moreover, we compare the results to (ii) Z3 1.2 [dMB08] and (iii) CVC3 1.2.1 [BT07]. The first column denotes the benchmark name. In the next column, labelled R, we show the number of refinements for our lemmas on demand approach. Columns T and M show solving time in seconds and memory usage in MB. Our approach clearly outperforms all the others.

Benchmark `swapmem` uses XOR in order to swap two byte sequences in memory twice. Extensionality is used to show that the final memory is equal to the initial. An instance is satisfiable if the sequences can overlap, and unsatisfiable otherwise. Benchmark `dubreva` reverses multiple byte sequences in memory twice. Again, extensionality is used to show that the final memory is equal to the initial. The instances are all unsatisfiable. In benchmarks `wchains` we check the following. Given P 32 bit pointers, the 32 bit word a pointer points to is overwritten with its pointer address. Then, the order in which the pointers are written does not matter as long the pointers are 4 byte aligned. Our benchmarks are part of the SMT library [BRST08].

As Boolector and STP have similar approaches we provide additional experiments comparing the performance of Boolector to the performance of STP. These results were published in the journal article [BB09d]. Note that we do not repeat the detailed results of SMT-COMP'08 [BDOS08], where Boolector clearly won the division of the quantifier-free theory of bit-vectors, arrays and uninterpreted functions `QF_AUFBV`, and the the di-

	lemmas			*eager*		z3		cvc3	
name	*R*	*T*	*M*	*T*	*M*	*T*	*M*	*T*	*M*
sm02s	16	0.1	0.6	**0.0**	**0.0**	0.1	5.0	554.1	18.2
sm06s	53	**0.4**	**1.2**	0.6	3.3	547.6	101.6	90.0	71.9
sm08s	37	**0.3**	**1.4**	1.5	5.4	out of time	out of time	333.8	175.8
sm10s	34	**0.5**	**1.7**	1.0	8.3	out of time	out of time	740.2	305.3
sm12s	64	**1.0**	**2.2**	3.5	11.2	out of time	out of time	out of time	out of time
sm14s	57	**1.6**	**2.4**	1.8	15.7	out of time	out of time	out of time	out of time
wc06s	14	**0.0**	**0.0**	0.8	7.0	46.3	11.6	8.5	60.1
wc08s	16	**0.0**	**0.0**	1.9	14.5	572.2	32.5	23.5	117.6
wc10s	18	**0.1**	**1.3**	3.1	20.3	out of time	out of time	34.2	193.6
wc12s	20	**0.1**	**1.4**	3.5	29.8	out of time	out of time	65.4	294.6
wc15s	21	**0.1**	**1.5**	4.2	45.8	out of time	out of time	126.3	472.6
wc20s	28	**0.3**	**2.0**	5.7	80.2	out of time	out of time	267.8	897.3
wc30s	36	**0.6**	**2.8**	10.9	179.9	out of time	out of time	out of memory	out of memory
wc60s	70	**2.3**	**4.8**	41.8	718.5	out of time	out of time	out of memory	out of memory
wc90s	96	**4.8**	**6.8**	out of memory	out of memory	out of time	out of time	out of memory	out of memory
dr02u	19	0.1	0.6	**0.0**	**0.0**	0.5	5.0	out of time	out of time
dr03u	41	0.3	**0.7**	**0.1**	1.2	0.7	5.6	out of time	out of time
dr04u	239	**12.5**	**2.2**	**12.5**	6.5	out of time	out of time	out of time	out of time
dr05u	379	**27.4**	**3.7**	348.3	13.1	out of time	out of time	out of time	out of time
dr06u	1187	**322.2**	**12.4**	out of time	out of time	out of time	out of time	out of time	out of time
dr07u	1637	**663.1**	**18.2**	out of time	out of time	out of time	out of time	out of memory	out of memory
sm02u	13	**0.1**	**0.7**	**0.1**	0.8	1.2	5.5	6.1	9.0
sm04u	25	**1.1**	**1.0**	3.0	2.0	5.2	7.1	159.6	49.8
sm06u	37	**3.2**	**1.2**	27.9	4.2	16.1	8.6	out of time	out of time
sm08u	49	**9.0**	**1.5**	129.0	8.3	20.7	10.1	out of time	out of time
sm10u	61	**18.3**	**2.0**	411.2	15.4	30.9	11.0	out of time	out of time
sm12u	73	**34.7**	**2.4**	out of time	out of time	62.1	13.3	out of time	out of time
sm14u	85	**63.4**	**2.8**	out of time	out of time	102.7	19.4	out of time	out of time
wc02u	17	**0.0**	**0.0**	**0.0**	**0.0**	out of time	out of time	4.1	5.8
wc04u	33	**0.1**	**0.9**	1.7	3.8	10.5	16.4	out of time	out of time
wc06u	49	**0.4**	**1.1**	13.5	8.2	1.9	5.5	out of memory	out of memory
wc08u	65	**0.6**	**1.3**	54.8	15.0	848.0	725.1	out of memory	out of memory
wc10u	81	**1.1**	**1.5**	158.2	23.3	59.7	14.6	out of memory	out of memory
wc12u	97	**2.4**	**1.8**	333.5	34.3	93.7	22.4	out of memory	out of memory
wc14u	113	**2.9**	**1.9**	588.2	46.7	19.3	6.9	out of memory	out of memory
wc16u	129	**4.8**	**2.1**	out of time	out of time	35.7	8.1	out of memory	out of memory
wc18u	145	**6.6**	**2.3**	out of time	out of time	209.8	26.2	out of memory	out of memory

Table 2.2: Experimental evaluation on extensional examples.

vision of the quantifier-free theory of bit-vectors `QF_BV`. Moreover, at SMT-COMP'09 [BDOS09], Boolector won in `QF_AUFBV` again, and was second best solver in `QF_BV`. We refer the reader to the webpage of the respective SMT competition [BDOS08, BDOS09] for more details.

STP does not support equalities between arrays. However, many interesting and challenging array benchmarks in the SMT-LIB [BRST08] compare arrays for equality, i.e. are extensional. Furthermore, STP reads the CVC Lite [BB04] format[4] while Boolector reads BTOR [BBL08] and SMT-LIB format [RT06]. This was problematic as formula conversion with the help of CVC3 was not always possible. Therefore, we had to restrict our experiments to non-extensional examples that were either available both in SMT-LIB and in CVC format, or were available in only one format and conversion to the other format was possible.

We selected the STP benchmarks that were available in `QF_AUFBV` in the SMT-LIB [BRST08]. Unfortunately, `testcase10` and `testcase15` were not available. We omitted two `cksumcookie` benchmarks as they were trivial for Boolector and STP. Furthermore, we selected seven benchmarks representing formal verification of the bubble sort algorithm for array elements of 32 bit. The number within the name of the benchmark represents the size of the array. These benchmarks are also part of the SMT-LIB and can be found in the division `QF_AUFBV`.

We ran our benchmarks on our cluster of 3 GHz Pentium IV with 2 GB main memory, running Ubuntu Linux. We set a time limit of 1800 seconds and a memory limit of 1500 MB. The results are shown in Tab. 2.3 and Tab. 2.4. The memory is shown in MB and the time in seconds. Boolector clearly outperforms STP[5].

[4] `http://verify.stanford.edu/CVCL/doc/`

[5] Note that STP can not decide satisfiability for benchmark `noregions-fullmemite`. After a few seconds it terminates with exit code 0, but does not print out the result. First, we thought that this might be a problem due to the YACC-parser. However, also CVC3 uses a YACC-parser and it can parse the benchmark without problems. Setting the stack size of the STP YACC-parser to unlimited did also not help. Therefore, it remains unclear why STP terminates without a result.

		STP		Boolector	
Benchmark	Status	Mem	Time	Mem	Time
610dd9dc	sat	out of memory	out of memory	**9**	**34**
blaster-concrete	unsat	48	2	**23**	**0**
blaster-small	sat	2	0	**0**	0
blaster	unsat	85	5	**31**	**0**
blaster-wp-13	unsat	48	2	**30**	**0**
blaster-wp-4	unsat	151	10	**38**	**1**
blaster-wp-8	unsat	68	4	**30**	**0**
bubsort005un	unsat	2	4	**1**	**0**
bubsort006un	unsat	3	30	**1**	**1**
bubsort007un	unsat	6	151	**1**	**8**
bubsort008un	unsat	11	784	**2**	**26**
bubsort009un	unsat	out of time	out of time	**2**	**72**
bubsort010un	unsat	out of time	out of time	**4**	**325**
bubsort012un	unsat	out of time	out of time	out of time	out of time
cmu-model15	sat	11	2	**5**	**0**
cmu-model16	sat	11	2	**5**	**0**
cmu-model17	sat	11	2	**4**	**0**
ff	unknown	out of memory	out of memory	out of memory	out of memory
grep0065	unsat	43	4	**2**	**0**
grep0084	sat	183	116	**18**	**15**
grep0095	sat	198	127	**19**	**17**
grep0106	sat	175	116	**20**	**18**
grep0117	sat	193	115	**20**	**19**
grep0777	sat	out of memory	out of memory	**33**	**99**

Table 2.3: Comparison between STP 0.1 and Boolector 0.8, part 1. The first column `Status` shows the satisfiability status of the instance. Columns `Mem` and `Time` shows memory consumption in MB and solving time in seconds.

		STP		Boolector	
Benchmark	Status	Mem	Time	Mem	Time
noregions-fullmemite	unknown	no result	no result	**380**	**40**
noregions-stpmem	unknown	out of time	out of time	**106**	**1526**
testcase01	sat	**22**	**2**	25	52
testcase02	sat	466	50	**413**	**14**
testcase03	sat	574	66	**500**	**18**
testcase04	sat	590	69	**514**	**18**
testcase05	sat	589	68	**513**	**18**
testcase06	unsat	**297**	31	490	**13**
testcase07	unsat	**298**	32	491	**13**
testcase08	unsat	**293**	31	490	**12**
testcase09	sat	584	68	**513**	**18**
testcase11	sat	**323**	15	515	**13**
testcase12	sat	**346**	16	550	**15**
testcase13	sat	**67**	5	79	**2**
testcase14	sat	63	28	**58**	**1**
testcase16	sat	952	**50**	**318**	75
testcase17	sat	347	23	**335**	**12**
testcase18	sat	43	**4**	**30**	87
testcase19	sat	28	3	28	3
testcase20	sat	**393**	**45**	505	117
testcase21	sat	**390**	**35**	520	53
thumbnail-spin1	sat	1116	75	**966**	**42**

Table 2.4: Comparison between STP 0.1 and Boolector 0.8, part 2. The first column `Status` shows the satisfiability status of the instance. Columns `Mem` and `Time` shows memory consumption in MB and solving time in seconds.

2.13 Related Work

Ganesh et al. present a decision procedure for the theory of bit-vectors and arrays [GD07], but without support for equalities on arrays. Similar to our approach, they use word-level preprocessing and an abstraction refinement loop. They implemented their decision procedure in STP, which is an SMT solver optimized for large problems in software analysis applications. Unfortunately, their decision procedure is presented without details. Neither in [Gan07] nor in [GD07] are details on their abstraction refinement and lazy axiom instantiations presented. Neither do they prove soundness nor completeness.

Ganesh points out in [Gan07] that efficiently handling nested writes is crucial for SMT solvers applied to software verification. An eager read-over-write elimination applied to deeply nested writes creates a new copy of the DAG of writes for every distinct read index. This eager rewriting blows up in space and is therefore not appropriate, which is also confirmed by our experiments in [BB08b]. Similar to our approach, Ganesh et al. handle read over writes lazily. Unfortunately, it remains unclear how their refinement actually works. In [Gan07] they give a hint that they use the following policy in their refinement. If they find a spurious model, then they take a non-constant array index term for which at least one array axiom is violated, and add all of the violated axioms involving that term. This is in contrast to our approach, where we always add exactly one lemma.

It is difficult to compare STP to Boolector as STP does not support equalities between arrays. Many interesting and challenging array benchmarks in the SMT-LIB [BRST08] compare arrays for equality. Therefore, we can not compare the performances of Boolector and STP on those examples. Nevertheless, our experiments on non-extensional examples, i.e. examples without comparing arrays for equality, in section 2.12 clearly show that Boolector outperforms STP, even on many examples for which STP has been optimized.

Stump et al. describe a decision procedure for an extensional theory of arrays in [SBDL01] and discuss earlier work. They present their decision procedure as a proof system and prove its correctness. Their approach works

in two phases. In the first phase a set of sub-goals is created such that no sub-goal contains write expressions, which may blow up in space. This is in contrast to our approach where we do *not* eagerly eliminate write expressions. Finally, the original goal is satisfiable if and only if one of the sub-goals is satisfiable. The key concept of their approach is to use partial equations of the following form:

$$a =_I b \quad \Leftrightarrow \quad \forall i.\ i \notin I \rightarrow \mathit{read}(a, i) = \mathit{read}(b, i)$$

The set I is a set of indices on which the arrays do not have to be equal. With these partial equations, write expressions are eliminated in the following way:

$$\mathit{write}(a, i, e) = b \quad \Leftrightarrow \quad a =_{\{i\}} b \ \wedge\ \mathit{read}(b, i) = e$$

Note that the idea of adding the constraint $\mathit{read}(b, i) = e$ to ensure write value consistency is similar to our preprocessing step 2. The complexity of their approach is $\mathcal{O}(2^{N\ lg\ N})$, where N is the size of the original goal.

Bradley et al. introduce the array property fragment in [BM07]. The main idea of this fragment is that array indices can be universally quantified with some restrictions. The restrictions are necessary to avoid undecidability. Array property fragments allow the expression of bounded and unbounded properties of arrays, e.g. array equality, universal properties, partitioning and sorting. Unlike our decision procedure, their decision procedure has to eliminate write expressions, which may blow up in space. Furthermore, the formula has to be put into negation normal form, which is also not necessary for our approach. Universally quantified sub-terms are reduced to finite conjunctions. This is done by instantiating quantified variables over a set of index terms. In [BMS06] Bradley et al. prove that simple extensions of the array property fragment, e.g. nested reads, may already result in a fragment for which satisfiability is undecidable. We conjecture that our approach can be extended to handle interesting properties of the array property fragment as well.

In [GNRZ07] Ghilardi et al. consider extensions of the theory of arrays with extensionality where indices have the algebraic structure of Presburger

arithmetic. Their extensions consider characteristics like dimension and sortedness. They integrate available decision procedures for the theory of arrays and Presburger arithmetic in combination with automatic instantiation strategies.

In [GKF08] Goel et al. introduce a rule-based axiom-instantiating decision procedure for the parametric theory of arrays, where formulas can refer to multiple and arbitrarily nested array types. Like our approach, they use lazy axiom instantiations. Interestingly, theories of arrays are interpreted in the context of updatable functions, i.e. *read* is interpreted as a function application and *write* as an update operator. They extended the congruence-closure module of the SMT solver DPT[6] in order to implement their approach. This is contrast to our approach, which is not based on congruence-closure. Unfortunately, the effectiveness of our implementation can not be compared to their implementation as they only report on experimental results in `QF_AUFLIA`, but not in `QF_AUFBV`.

Manolios et al. introduce a memory abstraction algorithm in the context of the Bit Level Analysis Tool BAT in [MSV06]. Memories are functionally represented with the help of array semantics, i.e. get and set in BAT correspond to read and write in SMT. Their approach uses equivalence classes of memories. Index values, i.e. bit-vector addresses that are used in set and get, are analyzed, and shorter bit-vector addresses for addressing the abstract memory are created. In addition to memory abstraction, rewriting is performed to simplify memory accesses.

In [BNO+08a] Bofill et al. introduce a write base approach in the context of DPLL(T) to handle the extensional theory of arrays. Their approach does not eliminate write expressions and does not lazily instantiate array axioms. However, the theory solver asks the boolean engine to split on demand on equality literals between indices.

The theorem prover Simplify [DNS05] uses Nelson-Oppen [NO79] to combine decision procedures for several theories, e.g. arrays, maps and sets, and uses a matcher to reason about quantifiers.

Finally, UCLID [LS04] is a decision procedure for the theories of uninter-

[6] `http://sourceforge.net/projects/dpt`

preted functions, integer linear arithmetic, and arrays which supports some limited forms of quantification. UCLID uses eager translation to SAT and lambda expressions for arrays. Furthermore, Bryant et al. propose a decision procedure for bit-vector arithmetic with automatic abstraction refinement in [BKO+09]. The procedure is implemented in UCLID. It alternates between under- and over-approximations of the original formula.

2.14 Conclusion

We discussed lemmas on demand for the extensional theory of arrays. In particular, we presented our preprocessing, formula abstraction, consistency checking, and abstraction refinement, i.e. lemma generation on demand. Furthermore, we proved that our approach is sound and complete, and provided a formal description of our abstraction refinement. We discussed complexity and provided implementation details and optimizations that are important for real implementations. Then, we discussed related work. Our overall experiments and the SMT competitions in 2008 and 2009 confirm that our approach implemented in our SMT solver Boolector is more efficient than other approaches in state-of-the-art solvers. Similar to model checking, where abstraction is used to handle the state explosion problem, abstraction techniques are the key to efficiently decide large and complex SMT formulas.

Chapter 3

Boolector

3.1 Introduction

Boolector is an efficient SMT solver for the quantifier-free theory of bit-vectors in combination with the extensional theory of arrays. Bit-vectors can be used to express designs and specifications directly on the word-level, while arrays can be used to model memory, e.g. main memory in software, or memory components like caches and FIFOs in hardware systems. The combination of bit-vectors and arrays allows reasoning about software and hardware on a more appropriate level than pure propositional logic. Generally, SMT solvers benefit from the additional word-level information and can generate word-level models. Boolector provides concrete models for bit-vectors *and* arrays.

The SMT competitions [BDOS08, BDOS09] in 2008 and 2009 showed, that there has been a lot of progress in the SMT community. In 2008, the winner of each division clearly outperformed last year's winner. In particular, the performance of SMT solvers in the quantifier-free theory of bit-vectors `QF_BV`, and bit-vectors with arrays and uninterpreted functions `QF_AUFBV`, increased heavily. Boolector participated in exactly these two divisions and won both. In `QF_BV` it solved 18 formulas more than Z3.2 and 92 formulas more than Spear v1.9 [Bab08] which was the winner of 2007. In `QF_AUFBV` Boolector solved 16 formulas more than Z3.2 and 64 more than Z3 0.1 [dMB08]

which was the winner of 2007. In the SMT competition [BDOS09] 2009, Boolector won again in `QF_AUFBV` and achieved the second place in `QF_BV`.

3.2 Architecture

Boolector depends on term rewriting and bit-blasting for bit-vectors. Lemmas on demand [dMR02] are used to handle the extensional theory of arrays lazily [BB09d]. It takes a formula expressed in the SMT-LIB format [RT06], or alternatively in the BTOR format [BBL08], as input. The BTOR format is a low-level bit-vector format with clean semantics that is easy to parse. Additionally, BTOR supports bit-vector arrays and model checking of safety properties. In addition to these input formats, Boolector also provides a rich C API, which allows to use Boolector as library. Boolector is implemented in C. A schematic overview is shown in Fig. 3.1. In the following we briefly discuss the main components and features of Boolector. After the overview, we discuss selected optimization techniques and features in detail.

Parser The parser reads the input and builds an abstract syntax DAG. During the parse process, basic rewriting rules are applied to simplify the DAG. Boolector can parse the SMT-LIB [RT06] and BTOR [BBL08] formats.

Rewriter The rewriting engine provides rewrite rules that can be divided into three levels. By default, all rewrite levels are applied to simplify the input. Level 1 rewrite rules are basic rules, e.g. $a \wedge \neg a \Leftrightarrow \bot$, that are applied during formula construction. Level 2 rewrite rules consist of global term substitutions in combination with static analysis techniques. Before terms are substituted, they are topologically sorted. Then, term substitutions are performed in one pass. Level 3 rewrite rules perform arithmetic normalization. Rewrite rules of quadratic worst case complexity are additionally bounded in recursion depth.

Array Consistency Checker This checker is one main component in the lemmas on demand approach for the quantifier-free extensional theory

of arrays [BB09d]. It checks if the current assignment by the SAT solver is consistent with the theory.

Under-approximation Boolector supports under-approximation of bit-vector variables and reads on bit-vector arrays. This module is responsible for realizing under-approximation directly on the CNF level. Under-approximation constraints are incrementally added as clauses that additionally constrain the search space. Boolector supports under-approximation techniques and refinement strategies. The under-approximation can be refined locally or globally. On the one hand, in the global refinement strategy, the under-approximation is refined equally for all variables and reads. On the other hand, in the local strategy, the under-approximation refinement is individually performed for each variable and read. The under-approximation feature allows to generate "smaller" models that are typically easier to interpret by users. We refer the reader to section 3.3 for a more detailed discussion of our under-approximation techniques.

SAT Solver PicoSAT [Bie08] is used as SAT solver. It uses state of the art techniques like watching literals, phase saving, conflict learning and rapid restarts. Boolector uses PicoSAT incrementally to decide the extensional theory of arrays, and for under-approximation.

Model Generator The ability to provide concrete models in the satisfiable case cannot be overestimated. In general, a satisfiable formula corresponds to a bug that has been found. The ability to provide a concrete counter example that can be directly used for debugging is one of the main features in the success story of model checking [Cla08]. Boolector can output concrete bit-vector *and* array models. The array models are (partially) instantiated arrays, where indices and values are concrete bit-vectors.

Formula Refinement The formula refinement is the heart of Boolector. Initially, the bit-vector part is translated to SAT while the array part is abstracted with the help of fresh bit-vector variables. The refinement loop

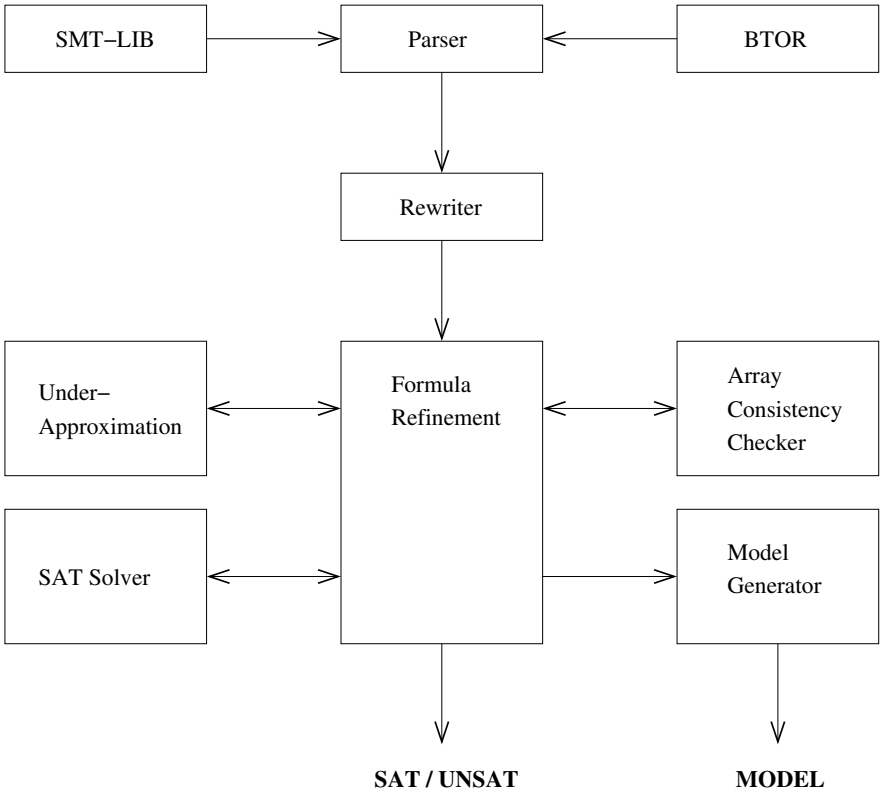

Figure 3.1: Schematic overview of Boolector.

calls the SAT solver to obtain an assignment. If the result is unsatisfiable, the loop terminates with *unsatisfiable*. However, if the result is satisfiable, the array consistency checker is used to check if the current assignment is consistent with the extensional theory of arrays. If the current assignment is consistent, then the formula is *satisfiable*. However, if the assignment is inconsistent, a lemma on demand, that rules out this and similar assignments, is added as formula refinement. Additionally, under-approximation refinement can be enabled for bit-vector variables and reads. In this case the under-approximation refinement and the lemmas on demand refinement for the extensional theory of arrays are intertwined. We refer the reader to section 3.3 for a discussion of the under-approximation techniques and to section 3.3.4 for a discussion of the intertwining technique.

Pretty Printer Boolector allows to convert formulas from BTOR to SMT-LIB format and vice versa. The pretty printer can be combined with Boolector's rewriting module to internally simplify the formula before conversion.

Model Checker Boolector can also be used as incremental model checker for word-level safety properties of synchronous hardware systems with memories [BBL08]. The BTOR format provides a sequential "next" operator, which can be used to express state transitions of bit-vector registers and memories. Boolector supports bounded model checking for witnesses, and k-induction with and without all-different constraints. All-different constraints are used for simple paths. Appendix A introduces the BTOR format and its next operator. Moreover, we provide experimental results on Boolector's model checking techniques.

Symbolic Overflow Detection Boolector directly supports symbolic overflow detection for fixed-size bit-vector addition, subtraction, multiplication and division in its interface. Moreover, symbolic overflow detection is supported in Boolector's native input format BTOR [BBL08]. An overflow detection operator can be used as a predicate that represents whether an overflow may occur or not. We refer the reader to section 3.5 for a discussion of our symbolic overflow detection techniques.

3.3 Under-Approximation Techniques

The main motivation of most approximation techniques is to speed-up decision procedures. While over-approximation technique tend to speed-up unsatisfiable formulas, under-approximation techniques tend to speed-up satisfiable formulas. In order to remain sound and complete, approximation techniques are typically combined with a refinement loop. Recently, approximation techniques are also used in the context of decision procedures for bit-vectors [BKO+09, HH08].

Over-approximation techniques are in the spirit of the Counter Example Guided Abstraction Refinement Framework [CGJ+03] (CEGAR) and are typically used in lazy SMT approaches [Seb07]. For example, over-approximation techniques are used to decide complex formulas containing arrays in [BB09d, Gan07]. In the rest of this section we will focus on under-approximation techniques.

The basic idea of under-approximation techniques in the context of bit-vectors is to restrict individual bits of bit-vectors. While such domain restrictions typically lead to a smaller search space and a speed-up for satisfiable formulas, it additionally produces "smaller" models which means that the domain of the variables are smaller. If a small model can be found, it must also be a model of the original formula. Small models are beneficial for diagnosis, for instance if the model is directly analyzed by users for debugging. Furthermore, in the area of test case generation, small models lead to test cases with reduced test data size.

One way of using over-approximations and under-approximations in bit-vector logic has been pioneered in [BKO+07, BKO+09]. In the context of under-approximation, the m most significant bits of variables are additionally restricted. The remaining n least significant bits are not concerned by the under-approximation and remain variable. In the rest of this section we call n the *effective bit-width*.

In [BKO+09] it is proposed to use an under-approximation technique which corresponds to sign-extension. Let n be the effective bit-width. The m most-significant bits of a variable are forced to be equal to the n^{th} least significant bit, the last effective bit. This technique reduces the domains of variables and leads to smaller models where bit-vectors are interpreted in the context of two's complement. An example with an effective bit width of four is shown left in Fig. 3.2.

It is also possible to force the m most significant bits to zero resp. one which we call *zero-extension* resp. *one-extension*. Zero-extension has been suggested in [BKO+09]. While zero-extension leads to smaller models where bit-vectors are interpreted in an unsigned context, one-extension is beneficial if a formula has small models with negative values. Examples with an

Figure 3.2: Under-approximation techniques: Sign-extension is shown left and zero-extension is shown right. The effective bit-width is four in both examples.

Figure 3.3: Under-approximation techniques: One-extension is shown left and class splitting is shown right. The effective bit-width resp. number of classes is four.

effective bit-width of four are shown in Fig. 3.2 resp. Fig. 3.3. In [HH07] zero-extension and one-extension were used for bounded model checking of embedded software.

We propose an additional under-approximation technique that *partitions bits* of individual bit-vectors into equivalence classes. All bits in one class are forced to have the same value. An example is shown Fig. 3.3. The under-approximation refinement increases the number of classes per variable, or splits individual classes. The idea of this technique is that only some individual bits of the vector are important to satisfy the formula. Therefore, the other bits can be forced to the same value which reduces the search space.

3.3.1 Under-Approximation on CNF Layer

We propose to perform under-approximations with the help of additional clauses on the CNF layer. The original formula is translated to CNF once. In each refinement iteration i, we perform an under-approximation by adding new clauses to the SAT solver incrementally and also use assumptions as presented in [CS03].

First, we introduce a fresh boolean under-approximation variable e. Then, we perform an under-approximation by adding new clauses. Let n be the effective bit width of a bit-vector variable v of bit-width w. To perform a

sign-extending under-approximation we add the following clauses:

$$\bigwedge_{i=n}^{w-1} ((v_{n-1} \vee \overline{v_i} \vee \bar{e}) \wedge (\overline{v_{n-1}} \vee v_i \vee \bar{e}))$$

Finally, we assume [CS03] e to enable the under-approximation.

For example, let v be a bit-vector variable with bit-width eight and an effective bit-width of six. We add the following clauses to perform a sign-extending under-approximation:

$$(v_5 \vee \overline{v_6} \vee \bar{e}) \wedge (\overline{v_5} \vee v_6 \vee \bar{e}) \wedge (v_5 \vee \overline{v_7} \vee \bar{e}) \wedge (\overline{v_5} \vee v_7 \vee \bar{e})$$

Assuming e enforces $v_5 = v_6 = v_7$.

Zero-extension can be encoded as follows. Again, let n be the effective bit width of a bit-vector variable of bit-width w. We add the following clauses:

$$\bigwedge_{i=n}^{w-1} (\overline{v_i} \vee \bar{e})$$

One-extension can be encoded analogously.

If we have to refine our approximation, we add the unit clause $\bar{e}$ in order to disable the current approximation. This gives the SAT solver also the opportunity to recycle clauses. Then, a refined under-approximation is performed with the help of another fresh boolean under-approximation variable.

3.3.2 Refinement Strategies

Generally, the effective bit-width can be used as a metric of approximation. Typically, the effective bit-width is initialized to one, i.e. the domain of a bit-vector variable is restricted to $\{-1, 0\}$ in a signed resp. $\{0, 1\}$ in an unsigned context. During the refinement the effective bit width is increased. In order to avoid too many refinement loops, the effective bit-width is typically doubled in each iteration. Traditionally, in the worst case, e.g. if the original formula is unsatisfiable, the effective bit-width reaches the original bit-width.

With the proposed refinement on the CNF layer, two main refinement

strategies are possible which we call *global* and *local*. On the one hand, local refinement strategies maintain one fresh boolean under-approximation variable e for each bit-vector in each refinement. The benefit is a precise refinement as we can ask the SAT solver if it has used the respective e to derive unsatisfiability. Only those under-approximations that have been used need to be refined. However, in the worst case we have to introduce $k \cdot r$ fresh variables, where k is the number of bit-vector variables and r is the maximum number of refinements.

On the other hand, the global refinement strategy maintains exactly one under-approximation variable for all bit-vector variables. The benefit is less overhead, as we need only r additional boolean variables, where r is the number of refinements. However, the refinement is imprecise.

3.3.3 Early Unsat Termination

Traditional under-approximation techniques perform the approximation outside the SAT solver. The CNF is generated from scratch in each refinement iteration. This makes it impossible to find out whether the current under-approximation has been responsible for deriving unsatisfiability or not. In the worst case, the original formula is unsatisfiable and we have the additional overhead of the under-approximation refinement, which is slower than solving the original formula up front.

The proposed under-approximation refinement on the CNF layer enables the decision procedure to terminate earlier, even if the original formula is unsatisfiable. If the under-approximated formula is unsatisfiable, then we can use the under-approximation variables to ask the SAT solver which under-approximations have been used to derive unsatisfiability. If no under-approximation variables have been used, then we can conclude that the original formula is unsatisfiable, and terminate. The early unsat technique is shown in Fig. 3.4.

Furthermore, the refinement on the CNF layer allows the SAT solver to keep learned conflict clauses over refinement iterations. This would be impossible if the CNF was generated on scratch in each refinement iteration.

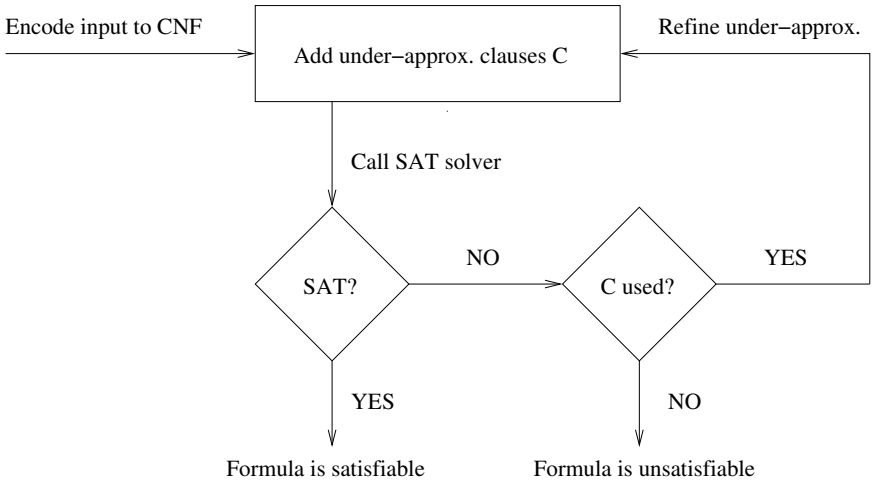

Figure 3.4: Early Unsat Termination.

In a first implementation we let the SAT solver generate unsat cores modulo assumptions, but simply recording those assumptions [CS03] that were used in deriving the empty clause is sufficient. Moreover, it is much faster, and easier to implement, both on the side of the SAT solver and on the side of the SMT solver.

3.3.4 Combining Approximation Techniques

Figure 3.5 shows how over-approximation and under-approximation techniques can be combined to solve complex SMT formulas. We consider the quantifier-free theory of arrays combined with the quantifier-free theory of bit-vectors. The idea is to use over-approximation techniques for the array part [BB09d, Gan07] and under-approximation techniques for bit-vectors.

First of all, we perform an over-approximation by replacing reads by fresh bit-vector variables. Then, we translate the bit-vector part of the formula to CNF and add a set C of under-approximation clauses. In each iteration we call the SAT solver. Depending on the result we have to perform an additional check. On the one hand, if the result is *satisfiable* we have to check if the current model σ respects the theory of arrays. If not, we have to refine our over-approximation with a lemma on demand [dMR02, FJS03, BDS02].

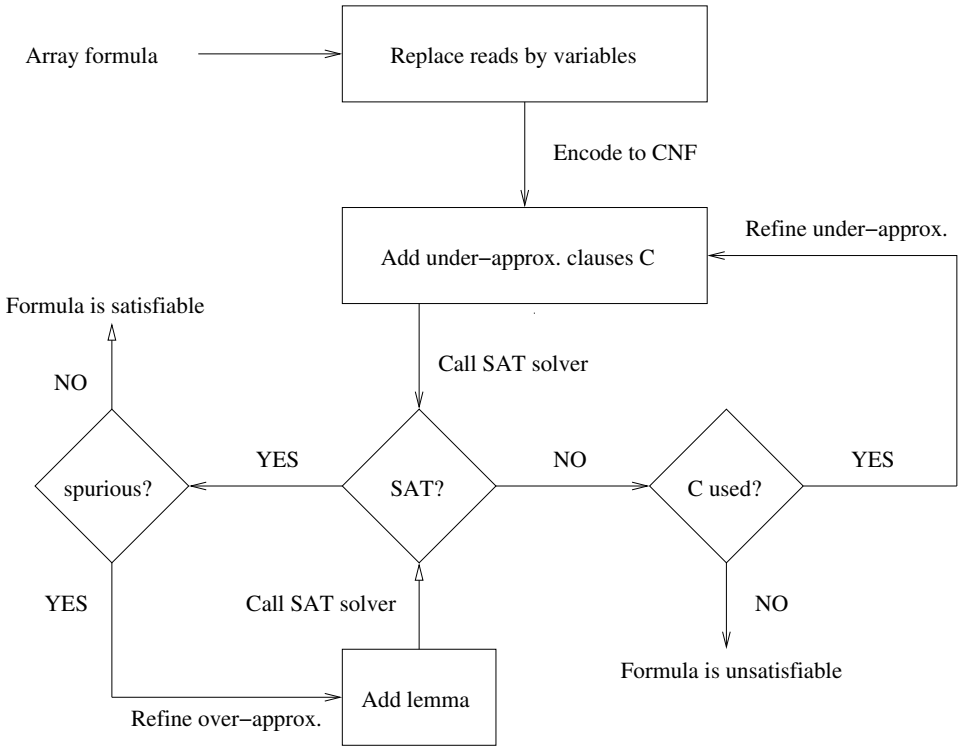

Figure 3.5: Combining over-approximation and under-approximation.

Otherwise, we can terminate with the model σ and the result *satisfiable*. On the other hand, if the result of the SAT solver is *unsatisfiable* we have to check if the current set of under-approximation clauses has been used. If not, we can terminate with the result *unsatisfiable*. Otherwise, we disable the current under-approximation and continue with a refined approximation.

3.3.5 Experiments

We implemented the presented approximation techniques in our SMT solver Boolector [BB09a]. Boolector is an SMT solver for the quantifier-free theory of fixed-size bit-vectors combined with the quantifier-free extensional theory of arrays [BB09d]. It is the winner of the last SMT competition in 2008 [BDOS08] in the bit-vector category (`QF_BV`) and also in the division of bit-vectors with arrays (`QF_AUFBV`). Moreover, Boolector can be used as

word-level bounded model checker for synchronous hardware and software systems [BBL08]. For our experiments we used Boolector 1.0.

The results of our experiments are shown in Fig. 3.6 and Fig. 3.7. We used benchmarks from `QF_AUFBF` of the SMT library [BRST08] (June 1st, 2008). As expected, under-approximation techniques speed up satisfiable instances, but slow down unsatisfiable instances. We summarize further observations.

First, the under-approximation by classes technique performs as good as the under-approximation by sign-extension technique on the satisfiable instances of `QF_AUFBF`. However, it performs worse on the unsatisfiable benchmarks. Second, the average ratio effective bit-width / original bit-width is 16% (without `egt` examples) on satisfiable and 27% on unsatisfiable benchmarks, which corresponds to an impressive reduction of 84% resp. 73%. Third, early unsat termination occurs in 1553 from 3552 unsat cases. Finally, the global refinement strategy seems to be a good approximation of the local strategy. It is much easier to implement, is often as good as the local strategy, and should be sufficient in most cases.

3.4 Unconstrained Variables

Independently from Roberto Bruttomesso in [Bru08], we observed that SMT benchmarks often contain *unconstrained* variables and implemented an optimization technique in our SMT solver Boolector. Unconstrained variables are inputs that are referenced only once in the formula, which means that they are not affected by further side conditions. Typically, their concrete value assignment is not essential for deciding the satisfiability of the formula. We can rewrite a formula containing unconstrained variables into a simpler equisatisfiable formula.

3.4.1 Bit-Vectors

In [Bru08] Roberto Bruttomesso showed that a bit-vector function f can be replaced by a fresh bit-vector variable if at least one of its operands is an unconstrained variable v, and f can be "inverted" with respect to v. We

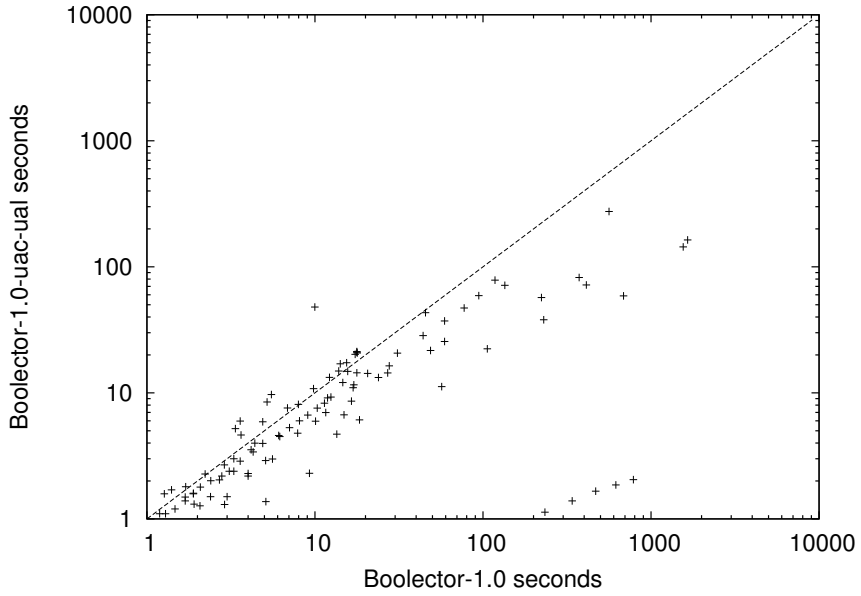

Figure 3.6: Boolector (x-axis) vs. Boolector with class under-approximation and local refinement strategy (y-axis). Benchmarks are from `QF_AUFBF` and are all satisfiable.

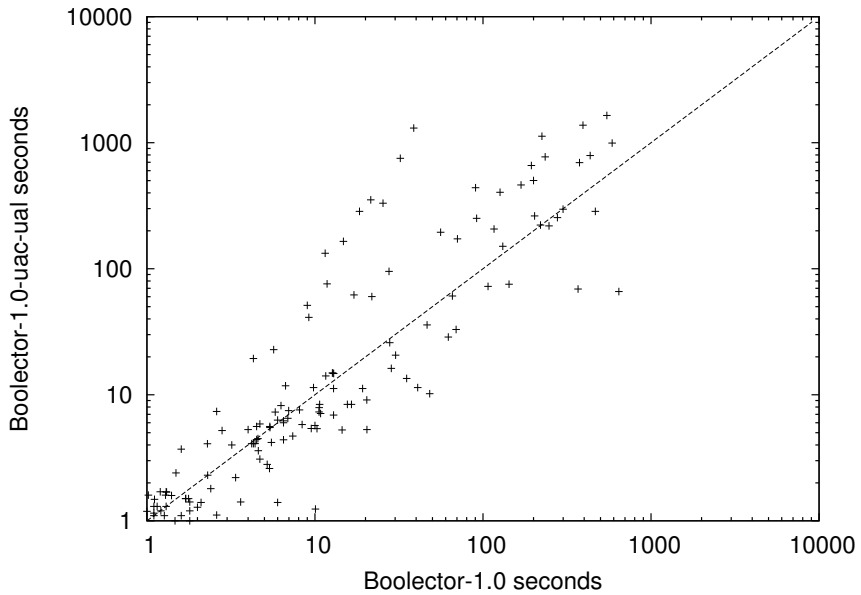

Figure 3.7: Boolector (x-axis) vs. Boolector with class under-approximation and local refinement strategy (y-axis). Benchmarks are from `QF_AUFBV` and are all unsatisfiable.

call a variable *unconstrained* in an SMT formula ϕ if it has only one parent in the abstract syntax DAG representation of ϕ. We assume that structural hashing is enabled in order to build the DAG representation.

For example, consider the following formula. Let t be an arbitrary bit-vector term, and let v_1, v_2 and v_3 be bit-vector variables. Moreover, assume that v_1, v_2, v_3 and t have bit-width 4, + denotes bit-vector addition, and & denotes bit-wise AND.

$$v_3 + t \quad = \quad v_1 \,\&\, v_2$$

Obviously, a decision procedure is free to choose the concrete assignments of v_1 and v_2 in order to obtain the concrete value of v_1 & v_2 that is needed to satisfy the formula. For example, if the decision procedure has fixed the left part of the equality to 0110, then v_1 & v_2 must also evaluate to this value in order to satisfy the formula. As v_1 and v_2 do not occur elsewhere in the formula, their assignments may be adjusted freely in order to satisfy $0110 = v_1$ & v_2, e.g. v_1 can be assigned to 1110 and v_2 can be assigned to 0111. As the assignments to v_1 and v_2 can always be adjusted in order to obtain the needed value for v_1 & v_2, we can replace v_1 & v_2 by a fresh bit-vector variable v_4 itself. In this way we can eliminate the bit-vector addition and obtain a simpler equisatisfiable formula. A similar argument can be applied to the left side of the equality. Independent from the assignment to t, the decision procedure can adjust the concrete assignment to v_3 to obtain the needed value for $v_3 + t$ in order to satisfy the formula. Therefore, we can replace $v_3 + t$ by a fresh bit-vector variable v_5 and obtain the following equisatisfiable formula:

$$v_4 \quad = \quad v_5$$

Note that t, which may be an arbitrary complex bit-vector term, has been completely eliminated.

In principle, it is straightforward to apply the same propagation technique to boolean operators, equality, if-then-else on terms, and bit-vector predicates. For example, it is obvious that we can replace $v_4 = v_5$ by a fresh boolean variable b in order to get a trivially satisfiable formula. In [Bru08] propagation rules are shown for addition, multiplication, not, and, or, con-

catenation, and if-then-else on terms.

There are some corner cases which must be considered in order to preserve correctness. For example, in [Bru08] it is shown that such a corner case is multiplication by an even constant. For example, if we consider the formula $110 \cdot v = 111$, then we are not allowed to replace the left side of the equality by a fresh bit-vector variable as this would result in a satisfiable formula, while the original formula is unsatisfiable.

Moreover, consider the following corner cases. Let $<$ denote the bit-vector predicate *unsigned-less-than*. Consider the formula $v < 000$. Obviously, we may not replace this formula by a fresh boolean variable as this would result in a satisfiable formula, while the original formula is unsatisfiable. Analogously, $111 < v$ is a corner case as well.

3.4.2 Arrays

We extend the optimization technique of propagating unconstrained variables to the quantifier-free extensional theory of arrays. Let a be an unconstrained array variable, i.e. it has exactly one parent. Recall that we assume that the original formula ϕ is represented as DAG. Moreover, let t be an arbitrary array term and v be an arbitrary unconstrained variable. Consider the following rewrite rules:

1. $\mathit{read}(a, i)$ is rewritten to a fresh variable of same sort.
2. $\mathit{write}(a, i, v)$ is rewritten to a fresh array variable of same sort.
3. $a = t$ is rewritten to a fresh boolean variable.

The first rewrite rule expresses that if a is an unconstrained array variable and $\mathit{read}(a, i)$ is the only operation applied to a, then the decision procedure can freely adjust the read value. Independent from the index i, which may be an arbitrary complex term, we can replace this read by a fresh variable. This rewriting eliminates the array variable a, the read $\mathit{read}(a, i)$, and possibly the index term i if it is referenced only at this place in the formula.

The second rewrite rule expresses that if we write an unconstrained value to an unconstrained array variable, then we can replace the result by a fresh

array variable. Interestingly, as in the read case, the index term i may be an arbitrary complex term as well. The main point of this rewrite rule is the following. Let j be an arbitrary index term. A decision procedure can freely adjust the value of $\mathit{read}(\mathit{write}(a, i, v), j)$. We have to consider two cases. On the one hand, if the current value of i is equal to j, then the read value has to be equal to the unconstrained variable v. As v is unconstrained, the read value can be freely adjusted by the decision procedure. On the other hand, if the current value of i is not equal to j, then the read is applied to a directly. As a is unconstrained, the read value can be freely adjusted by the decision procedure. Obviously, applying reads to $\mathit{write}(a, i, v)$, where a and v are unconstrained, is equisatisfiably equal to applying reads to an unconstrained array variable. Therefore, we can replace $\mathit{write}(a, i, v)$ by a fresh array variable a in order to obtain a simpler and equisatisfiable formula.

The third rewrite rule expresses that if we compare an unconstrained array variable with an arbitrary array term for equality, then we can replace the equality by a fresh boolean variable to obtain a simpler and equisatisfiable formula. As a is unconstrained, the decision procedure can either assign the unassigned elements of a and t in such a way that a and t are extensionally equal, or it can generate an assignment such that the two arrays differ at one particular position k, i.e. the concrete value of $\mathit{read}(a, k)$ is not equal to $\mathit{read}(t, k)$.

As an example, consider the following formula. Let $a_1 \cdots a_3$ be arbitrary unconstrained array variables, $i_1 \cdots i_5$ arbitrary index terms, and $v_1 \cdots v_3$ arbitrary unconstrained value terms.

$$\mathit{read}(\mathit{write}(\mathit{write}(a_1, i_1, v_1), i_2, v_2), i_3) = \mathit{read}(a_2, i_4) \ \wedge \ \mathit{write}(a_3, i_5, v_3) = a_3$$

Starting from a_1, we use rewrite rule 2 recursively to rewrite the array term $\mathit{write}(\mathit{write}(a_1, i_1, v_1), i_2, v_2)$ into a fresh unconstrained array variable. As the only parent of this new array variable is the read at index i_3, the array variable is unconstrained as well. Therefore, we can perform rewrite rule 1 to rewrite the read into a fresh variable v_4. The other read $\mathit{read}(a_2, i_4)$ can be directly rewritten into v_5 by rule 1. Now, we apply rule 3 to rewrite

$write(a_3, i_5, v_3) = a_3$ into a fresh boolean variable b_1 in order to obtain the following intermediate result.

$$v_4 = v_5 \ \wedge \ b_1$$

Obviously, we can rewrite $v_4 = v_5$ into a fresh boolean variable b_2 and obtain $b_2 \ \wedge \ b_1$. Again, we can rewrite this formula into the fresh boolean variable b_3 which is trivially satisfiable.

3.4.3 If-then-else

Finally, we discuss how unconstrained variables can be propagated through if-then-else on terms. Let p be an arbitrary formula, and let v_1 and v_2 be arbitrary terms of the same sort. As already pointed out in [Bru08] for bit-vectors, we can rewrite a sub-term $ite(p, v_1, v_2)$ into a fresh variable of the same sort as v_1 and v_2 to obtain an equisatisfiable formula. The main observation is, that the result of *ite* is always an unconstrained variable, regardless of the assignment to p.

We introduce another rewrite rule for if-then-else on terms. Let b be an unconstrained boolean variable, let v be an unconstrained term variable, and let t be an arbitrary term. Moreover, v and t are of the same sort. We can rewrite the term $ite(b, v, t)$ into a fresh variable of the same sort as v and t. Analogously, we can rewrite $ite(b, t, v)$. A decision procedure is free to assign the appropriate value to b in order to select the unconstrained variable v. Therefore, we can rewrite the if-then-else in order to obtain a simpler equisatisfiable formula.

3.5 Symbolic Overflow Detection

Overflow detection plays an important role in verification, in particular in the context of software verification. To avoid growth in bit-vector length, the resulting bit-width of arithmetic operations such as fixed-size bit-vector addition is the same as the bit-width of the inputs. This is a well-known

source of software defects as the result is not fully representable in general.

In contrast to other SMT solvers, Boolector supports symbolic overflow detection for fixed-size bit-vector addition, subtraction, multiplication and division directly in its interface, and also in its native input format BTOR [BBL08]. An overflow operator is simply a predicate that returns whether an overflow may occur or not. Such overflow predicates can be collected and integrated into verification conditions in order to explicitly consider verification in the context of overflows.

In the following we discuss how Boolector's symbolic overflow detection operators are implemented on the term layer. Direct support of overflow detection takes the load off the user, i.e. the user does not have to explicitly build non-trivial and error prone overflow detection circuits for each bit-vector operation where overflow detection should be considered.

In the following we assume that the two's complement is used to represent signed bit-vectors. With $a[p]$ we denote the bit of bit-vector a with bit-width n at position p, with $0 \leq p < n$. We use the symbols a, b and r to denote bit-vectors, n to denote a fixed bit-width with $n > 0$, and $c[p]$ to denote the carry-out of the full-adder at position p with $0 \leq p < n$. In the context of two's complement semantics, we write $S(a)$ to denote the sign-bit of a, i.e. the most significant bit of the fixed-size bit-vector a.

Moreover, we write $Z(a)$ to denote the number of leading zeros of a, and $L(a)$ to denote the number of leading bits of a. The number of leading bits is either the number of leading ones for bit-vectors that represent negative integers, or the number of leading zeros for bit-vectors that represent positive integers. For example, the bit-vector constant 11100101 has four leading bits, while 00101010 has two leading bits. Finally, in order to keep the presentation short, we sometimes use $a[p]$ as abbreviation for the truth value of $a[p] = 1$.

3.5.1 Addition

Unsigned overflow detection

Boolector uses a ripple-carry-adder implementation for bit-vector addition, represented as vector of AIGs. Bit-vector addition overflows can be simply

detected by examining the carry-out of the full-adder at the most significant bit. An overflow occurs if and only if $c[n-1]$ is 1 [Wir95].

Unsigned bit-vector addition overflow detection is implemented in Boolector as follows. We have to examine $c[n-1]$ in order to detect if an overflow may occur or not. However, we cannot directly examine $c[n-1]$ as it is not part of the resulting bit vector, i.e. the carry-out of the full-adder at the most-significant bit position is discarded. Therefore, we have to extend both operators by a leading zero to perform a bit-vector addition of bit-width $n+1$, i.e. we compute $r = \{1'b0, a\} + \{1'b0, b\}$. Finally, we are able to examine $r[n]$ which is equal to $c[n-1]$ in order to determine if an overflow may occur.

Signed overflow detection

In the context of signed bit-vector addition, an overflow occurs if and only if the operands have equal sign bits, but the sign bit of the result differs. This fact can be expressed more formally in the following way. Let $r = a + b$ be the resulting bit-vector of bit-width n. An overflow occurs if and only if

$$(S(a) \wedge S(b) \wedge \neg S(r)) \vee (\neg S(a) \wedge \neg S(b) \wedge S(r))$$

Note that signed addition overflows can also be detected by computing $c[n-1] \oplus c[n-2]$ [Wir95], where $\oplus$ denotes XOR. However, this principle is not suitable for Boolector as its overflow detection is directly implemented on the term layer where individual carry-out bits of the underlying AIG layer cannot be accessed easily.

3.5.2 Subtraction

Unsigned overflow detection

As usual, Boolector implements bit-vector subtraction $a - b$ as $a + (-b)$. An overflow occurs if and only if $c[n-1]$ is 0.

Signed overflow detection

For signed bit-vector subtraction there are exactly two cases where an overflow may occur. In the following we interpret the bit-vector operands as integers:

- We subtract a negative integer from a positive integer and the result is negative.
- We subtract a positive integer from a negative integer and the result is positive.

Let $r = a - b$ be the resulting bit vector of length n. For signed bit-vector subtraction an overflow occurs if and only if $(\neg S(a) \wedge S(b) \wedge S(r)) \vee (S(a) \wedge \neg S(b) \wedge \neg S(r))$.

3.5.3 Multiplication

Simple multiplication overflow detection circuits compute a $2n$ product and examine the n most significant bits of the result. This approach is not feasible for symbolic overflow detection as symbolic multiplication circuits are very challenging for decision procedures. Moreover, the n most significant bits are not needed for any other computations and are therefore not shared by structural hashing techniques.

In [SGBB00], overflow detection is implemented by combining several overflow detection units. Moreover, only a $n+1$ product has to be computed instead of a $2n$ product. Inspired by this technique, we implemented overflow detection directly on the term layer as follows.

Unsigned overflow detection

Let $r = a * b$ the resulting bit vector of length $n+1$. Recall that $Z(a)$ denotes the number of leading zeros for bit-vector a. An overflow occurs if and only if $Z_a + Z_b < n - 1$ or $r[n]$ is 1 [SGBB00].

Figure 3.8 shows the principle how Boolector detects if the sum of leading zeros is less than $n - 1$ or not. First, we examine the most significant bit

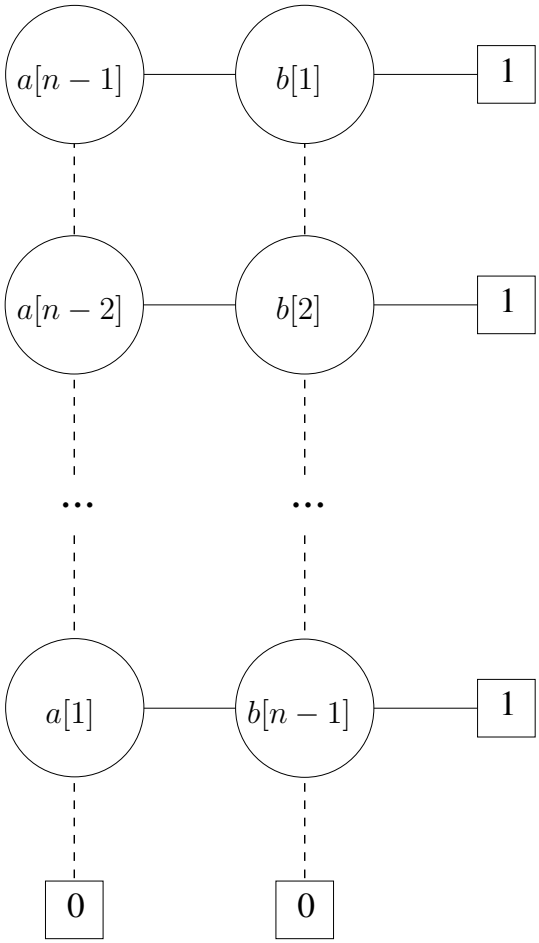

Figure 3.8: Leading zeros examination principle. The examination starts at $a[n-1]$ and moves either right or down. Dashed edges are taken if the considered bit is zero, and solid edges otherwise.

$a[n-1]$. If this bit is 0, then we have found one leading zero and we continue our examination with $a[n-2]$. However, if $a[n-1]$ is 1, then a does not have any leading zeros at all and we examine $b[1]$. If $b[1]$ is 1, then b can have at most $n-2$ leading zeros and we have detected an overflow. However, if $b[1]$ is zero, then we continue our examination with $b[2]$.

If we reach the constant 1 in the graph, then we know that the sum of leading zeros is less than $n-1$ and we have detected an overflow. However, if we reach the constant 0, then we know that the sum of leading zeros is greater than or equal $n-1$. If this is the case, we have to examine $r[n]$ in order to determine if an overflow may occur or not.

This examination principle leads to the following formula which is used by Boolector internally to determine if the sum of leading zeros is less than $n-1$ or not.

$$\bigvee_{i=1}^{n-1} (a[i] \wedge \bigvee_{j=1}^{i} b[n-j])$$

For example $(a[1] \wedge b[3]) \vee (a[2] \wedge (b[3] \vee b[2])) \vee (a[3] \wedge (b[3] \vee b[2] \vee b[1]))$ determines for $n = 4$ if the number of leading zeros is less than $n-1$ or not.

Signed overflow detection

$r = a * b$ the resulting bit vector of length $n+1$. Recall that $L(a)$ denotes the number of leading bits for bit-vector a. For signed multiplication an overflow occurs if and only if $L(a) + L(b) < n$ or $r[n] \oplus r[n-1]$ [SGBB00].

The number of leading bits can be detected as follows. We compute a' and b' where each bit-vector is logically XOR-ed with the sign bit. For example, if $a = 11100101$, then a' is 00011010. As a result of this computation a' and b' have now leading zeros only, so we can use almost the same examination principle as in the unsigned case. Figure 3.9 shows the principle how Boolector detects if the number of leading bits is less than n or not.

Again, if we reach the constant 1 in the graph, then we know that the sum of leading bits is less than n and we have detected an overflow. However, if we reach the constant 0, then we know that the sum of leading bits is greater than or equal n. In this case, we have to examine $r[n] \oplus r[n-1]$ in order to

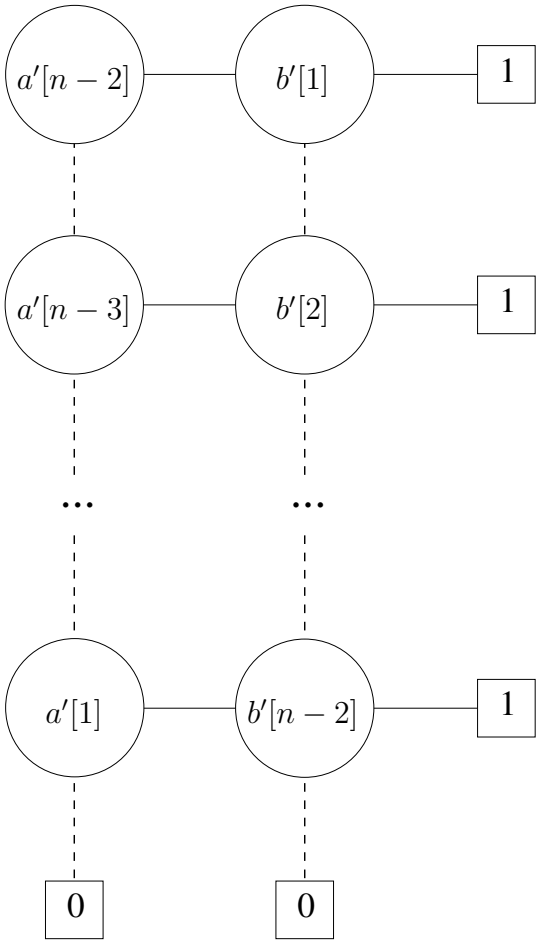

Figure 3.9: Leading bits examination principle. The examination starts at $a[n-2]$ and moves either right or down. Dashed edges are taken if the considered bit is zero, and solid edges otherwise.

determine if an overflow may occur or not.

This examination principle leads to the following formula which is internally used by Boolector to determine if the sum of leading bits is less than n or not.

$$\bigvee_{i=1}^{n-2} (a'[i] \wedge \bigvee_{j=1}^{i} b'[n-j-1])$$

For example $(a'[1] \wedge b'[2]) \vee (a'[2] \wedge (b'[2] \vee b'[1]))$ determines for $n = 4$ if the number of leading bits is less than n or not.

3.5.4 Division

Let a and b be two bit vectors of length n. In the context of signed division, there is exactly one case where an overflow may occur. An overflow occurs if and only if the smallest representable integer value is divided by -1. Note that there is no case where an overflow occurs for unsigned division.

3.6 Conclusion

We presented Boolector, which is an efficient SMT solver for the quantifier-free theories of bit-vectors and arrays. Boolector uses term rewriting, bit-blasting for bit-vectors, and lemmas on demand for arrays. We discussed its architecture, main concepts, implementation and optimization details, and selected features. In particular, we discussed an optimization technique which propagates unconstrained bit-vector and array variables, symbolic overflow detection, and novel under-approximation refinement techniques that work directly on the CNF layer.

Chapter 4

Testing and Debugging SMT Solvers

SMT solvers are widely used as core engines in many applications. Therefore, robustness and correctness are essential criteria. Current testing techniques used by developers of SMT solvers do not satisfy the high demand for correct and robust solvers, as our testing experiments show. To improve this situation, we propose to complement traditional testing techniques with grammar-based blackbox fuzz testing, combined with delta-debugging. We demonstrate the effectiveness of our approach and report on critical bugs and incorrect results which we found in current state-of-the-art SMT solvers for bit-vectors and arrays.

4.1 Introduction

Many applications use Satisfiability Modulo Theories (SMT) solvers as core decision engines. For example, SMT solvers are used to generate test cases, to find bugs [BTV09, CDE08, CGP+06, TdH08, VTP09], and to verify systems [Bab08, BBC+06, HHK08, SJPS08, JES07, LQ08]. A crashing SMT solver may lead to a crash of the application, or even worse, an incorrect solver may lead to wrong results. For example, if an SMT solver concludes *unsat* although the input formula is *sat*, a verification system may spuriously

conclude that an implementation respects its specification, i.e. defects are missed.

We show that although there is a high demand for robustness and correctness of SMT solvers, many state-of-the-art solvers were broken at the time of our tests. They contained defects that led to crashes, or even worse, to incorrect results where the solver concludes *sat* although the formula is *unsat*, or vice versa. We demonstrate that many critical defects can be found by a testing technique called *grammar-based blackbox fuzz testing*, and propose to complement traditional testing approaches with this technique. Moreover, we propose to integrate *delta-debugging* [Zel05] into the debugging process in order to minimize failure-inducing SMT formulas.

4.2 Fuzzing

Fuzzing is a powerful testing technique which is typically used in the domains of software security and quality assurance [SGA07, TDM08]. The main idea of the original fuzzing approach is to test programs with random inputs in order to detect security bugs, e.g. buffer overflows. Fuzz testing techniques were already applied by software engineers around 1980. For example, a tool called "the monkey" was developed to test the original Macintosh system. It fed random events to the current application. It appeared as if the computer was operated by an angry monkey [TDM08]. In [MKL+95], fuzz testing was used to find bugs in UNIX tools.

The first tools were simple, used *blackbox* testing, i.e. testing was performed against the interface without access to implementation details, and had no knowledge of the expected input format. In the context of SMT solvers we are typically not interested in fuzzing techniques that are unaware of the input syntax. Although useful, such techniques would mainly test syntax error handling routines of the parser. We would only scratch the surface and would not be able to test deeper parts of the solver. However, we focus on finding critical bugs such as crashes upon syntactically valid inputs, and incorrect results.

Nowadays, various fuzzing tools are available [SGA07, TDM08]. More-

over, there is an ongoing research on *whitebox* fuzz testing tools, which are a new generation of sophisticated fuzzers that use recent advances in symbolic execution and dynamic test generation [GKL08, GLM08]. Whitebox fuzzing is an interesting application for SMT solvers. However, we use fuzzing techniques in order to *test* solvers. Although the *whitebox* fuzzing approach seems promising for testing solvers, it has its limitations when it is used for programs that expect highly structured inputs such as the SMT format [RT06]. Sophisticated and complicated techniques like *grammar-based whitebox fuzzing* [GKL08] are necessary in order to use *whitebox* techniques for testing deep parts of SMT solvers, e.g. error prone optimizations.

We propose to use *grammar-based blackbox fuzzing* in order to test SMT solvers. A fuzzer randomly generates syntactically valid SMT formulas in order to detect critical defects. Unlike *grammar-based whitebox fuzzing*, *grammar-based blackbox fuzzing* is easy to implement and integrate, generates no false positives, and is impressingly effective in finding bugs that traditional testing techniques miss. This is confirmed by our experiments in section 4.4. Moreover, in contrast to fuzzing techniques such as [CH00], it does not need any user specifications and can be fully automated.

4.2.1 Generating Random Bit-Vector Formulas

We use a layered approach for generating random formulas, similar to [Vid08]. In the following we focus on bit-vector formulas. However, the main concepts and ideas can also be used in the context of other theories.

Although the main algorithm is rather simple, many details have to be considered as the quantifier-free theory of fixed-size bit-vectors has many different operators. While some of them use bit-vector arguments only, other operators, e.g. `extract`, also use integer constants. Moreover, the operators require different preconditions, e.g. equal bit-width of the operands, valid index positions, etc., which have to be fulfilled. In contrast to [Vid08], we also have to consider more types, i.e. unlike the BTOR format [BBL08], the SMT-LIB format [RT06] distinguishes between type *boolean* and *bit-vector of bit-width one*.

In principle, we structure a bit-vector formula into four layers: *input*, *main*, *predicate*, and *boolean*. During the formula construction we maintain a set of all nodes that have been created, including their types.

First, we generate the *input* layer with a random number of bit-vector variables and constants. The bit-width is also selected randomly which makes generating random bit-vector formulas more complicated than formulas that contain natural numbers. We have to consider many different types as we do not want to restrict all bit-vector terms to the same bit-width.

Second, we generate the *main* layer. We iteratively choose either a random bit-vector operator[1], or one of {`=`, `distinct`, `ite`}. Then, depending on the arity of the operator, we randomly select operands from our set, generate the final node, and insert it into the set. The main problem in this step is that the operands might have different bit-widths, but almost every bit-vector operator requires them to be equal. We use the extension operators `zero_extend` and `sign_extend`, and the extraction operator `extract` to solve this problem. If the operands do not have the same bit-width, then we either extend the "smaller" operand, or randomly select one possible sub-vector of the "larger" operand.

In the *main* layer we are interested in bit-vector nodes only. However, some operators, i.e. `=`, `distinct`, and bit-vector predicates, result in boolean nodes. In order to convert them to bit-vector terms we use an if-then-else wrapper. Assume t_1 and t_2 are bit-vector terms with the same bit-width, and p is a bit-vector predicate. We convert the predicate $p(t_1, t_2)$ into a bit-vector term as follows:

$$(ite\ (p\ t_1\ t_2)\ bv1[1]\ bv0[1])$$

If $p(t_1, t_2)$ is true, we return the bit-vector constant one with bit-width one, and zero otherwise. Hence, we can use wrapped predicates as inputs to bit-vector operations. This technique aims to detect subtle bugs that are not found by tests that use predicates in a boolean context only.

Third, we analogously generate the *predicate* layer. However, we restrict our operator selection to `=`, `distinct`, and bit-vector predicates. Finally, we

[1] In the SMT-LIB there are currently 35 bit-vector operators in `QF_BV`.

generate the *boolean* layer by randomly combining boolean nodes to one final root. We iteratively select roots, i.e. boolean nodes without parents, from our boolean layer, and combine them by a random boolean operator. We continue this process until there is only one root left.

4.2.2 Bit-vector Arrays

Adding one-dimensional bit-vector arrays is straightforward. We extend our algorithm for generating random bit-vector formulas as follows. First, we add array variables to the *input* layer. Then, during the process of building the *main* layer, we also build an *array* layer by using $write(a, i, e)$, where a is an array, i a bit-vector index and e a bit-vector value. The result of $write(a, i, e)$ is an *array* where the value at position i has been overwritten by e. All other elements of a remain the same. We select a, i, and e randomly from the terms that have already been created. If the bit-widths of i and e are incompatible to a, we use the techniques described earlier.

Moreover, while we are building the *main* bit-vector and array layer we also create reads by using $read(a, i)$, where a is an array and i is a bit-vector index. Analogously to $write(a, i, e)$, we select a and i randomly from the terms that have been created before, and either extend or slice i if necessary.

Interleaving the phases of creating regular bit-vector terms, reads and writes ensures that reads are also used as read indices, write indices, write elements, and also as operands of regular bit-vector operations. Moreover, nested writes may be created. Generating such formula structures aims to find subtle bugs in array algorithms that use abstraction refinement loops such as [BB09a, Gan07] where reads are internally replaced by fresh variables.

If we want to support the extensional theory of arrays where we can compare arrays in addition to array elements, then we can extend the *main* bit-vector layer with array equalities, encoded as bit-vectors. Moreover, array equalities may also be added to the *boolean* layer.

4.3 Delta-Debugging SMT Formulas

After we have found failure-inducing inputs, we typically want to minimize them. Delta-debugging techniques [Art08, CH00, MS06, Zel05, ZH02] *automatically* simplify and isolate failure-inducing inputs by using a divide-and-conquer strategy. Typically, minimized inputs speed up debugging as non-irrelevant input parts do not have to be considered. Moreover, large inputs may be practically infeasible to debug.

A delta-debugger repeatedly calls the program with simplified variants of the failure inducing input. If the program shows the same observable behavior, e.g. returns the same exit code or prints out the same error message, the delta-debugger continues with the simplified input, and backtracks otherwise.

Generally, delta-debugging does not generate a minimal failure-inducing input. However, this feature is rarely needed in practice. Typically, the goal of delta-debugging is to reduce the input as much and as fast as possible. It is not feasible for engineers to wait for a delta-debugging tool that needs days or even weeks to terminate. Therefore, we use a small and simple set of simplifications which allows fast delta-debugging while generating very small failure-inducing inputs. This is also confirmed by our experiments in Tab. 4.3 and Tab. 4.4.

In the context of SMT we have highly structured inputs, and type information. As Zeller's original delta-debugging technique [Zel05] does not explicitly use any knowledge of the input structure, we use a variant of hierarchical delta-debugging [MS06]. Hierarchical delta-debugging techniques have also been proposed for BTOR [Vid08]. We use the knowledge of formula structures and types to speed up the delta-debugging process, and to minimize inputs even further.

First of all, we represent an SMT formula as DAG with one boolean root. Typically, SMT formulas are layered, e.g. there is a boolean layer on top of the formula. We use the knowledge of a boolean layer to prune large irrelevant parts of the input up front. We iteratively try to replace the current root by one of its boolean children, i.e. we perform an eager and greedy search through the boolean layer.

Then, we perform further term-level simplifications, driven by a breath-first-search. Whenever new nodes are created, we insert them into a queue for further simplifications. We try to substitute each node either by the constant zero, one, or by one of its children, but only if the types of the current node and its child match. Note that this is not possible for bit-vector predicates, as they have a boolean type while their children have bit-vector types. If one child is a chain of unary operator applications, or writes, we try to skip it, i.e. we try to replace the current node by the node at the end of the chain. By using this technique we may immediately prune irrelevant operator chains, e.g. deeply nested writes.

Finally, after all nodes have been processed, we try to find a new root again. Boolean nodes may not only occur within the *boolean* layer, but also deeper inside the formula, used as conditions in if-then-else operators. We expect that the input formula has now been simplified significantly by the delta-debugging process. Therefore, we can try to substitute the current root by arbitrary boolean nodes that occur deeper in the formula. In this way we may minimize the formula even further and eliminate irrelevant if-then-else nodes in upper formula layers.

4.3.1 Delta-Debugging Crashes

Typically, fuzz testing leads to a high number of system crashes, i.e. the program terminates without providing a result. This is confirmed by our experiments in section 4.4. The randomness of the input is responsible for triggering statements that have not been tested before. Executing untested statements may lead to erroneous internal states, and thus, crashes.

In the context of SMT solvers, we have observed that crashes typically occur almost immediately after starting a solver. We conjecture that this observation also holds for other kinds of solvers that use complex and error prone optimization techniques, e.g. SAT solvers. This is confirmed by preliminary internal experiments on fuzzing SAT solvers. We can use the "early-crash" observation to improve the performance of delta-debugging.

First, we introduce the concept of timeouts. During delta-debugging,

each call to the solver is executed using a time limit. Whenever the solver exceeds its limit, we treat this case as if the simplification of the failure-inducing input has failed, and backtrack. Note that using timeouts during delta-debugging can also be useful for other kind of defects. For example, it can be used for delta-debugging failure-inducing formulas that lead to an infinite loop within the solver. We refer the reader to our experiments in section 4.4.2 where we provide a discussion of delta-debugging SMT solvers that do not always terminate.

Whenever we want to delta-debug a formula that leads to a crash, we typically set the time limit to a few seconds above the time which leads to a crash on the original formula. Using timeouts may heavily speed up delta-debugging formulas that are hard to decide. For example, assume we want to delta-debug a complex SMT formula. Whenever a specific sub-formula structure occurs, the solver crashes almost immediately. However, whenever delta-debugging simplifications destroy this sub-formula structure, the solver works correctly and may run for days. By using timeouts we can backtrack simplifications that do not lead to a crash almost immediately, and thus, speed up delta-debugging significantly, i.e. the delta-debugger may terminate with a small failure-inducing input already within a few minutes.

4.4 Experiments

4.4.1 Bit-Vector Theories

To evaluate the effectiveness of our approach, we fuzz-tested and delta-debugged publicly available state-of-the-art SMT solvers with our fuzzer `FuzzSMTBV` and our delta-debugger `DeltaSMT`. The failure-inducing inputs and delta-debugged results are publicly available[2]. Our SMT Tools, including `FuzzSMTBV` and `DeltaSMT` are publicly available as well[3].

We ran our experiments under Ubuntu Linux on an Intel Core 2 Quad machine with 2.66 GHz and 8 GB RAM. Our fuzz testing framework used

[2] `www.fmv.jku.at/brummayer/fuzz-dd-smt.tar.7z`
[3] `www.fmv.jku.at/software/index.html#smttools`

each of the four cores for testing. Our delta-debugging experiments were performed on the same machine, but were not run simultaneously.

The process of testing SMT solvers was rather complicated and complex. At the time of our tests, only two solvers, Boolector [BB09a] and Z3 [dMB08], supported all bit-vector operators of the SMT-LIB [BRST08] without crashing. Therefore, we used Boolector as a filter to rewrite high-level operators, e.g. smod, into a combination of supported low-level base operators according to [BBL08]. Although we had to restrict the tests to at most 11 out of 35 bit-vector operators, we found an impressive number of bugs. We conjecture that we would have found even more bugs if we had been able to use the full set of operators. This is also confirmed by further unpublished experiments. In Appendix B we list the most interesting error messages that we encountered during our fuzzing experiments.

For the quantifier-free theory of bit-vectors `QF_BV` we tested the following solvers: Beaver [JLS09] 1.1-RC1, Boolector [BB09a] 1.0 and 1.1, a development version of CVC3 [BT07] 1.5 (downloaded April 29th, 2009), MathSAT [BCF+07] 4.2.3, Spear [Bab08] 2.7 with SMT2SF 1-9, Sword [WFG+07] from SMT-COMP'08 [BDOS08] and an unstable version of OpenSMT [BS08]. Moreover, we tested Z3 [dMB08] 1.2 and Z3 from SMT-COMP'08 [BDOS08]. The results are summarized in Tab. 4.1.

Note, the current versions of CVC3 and OpenSMT do not support bit-vector division. Moreover, we could not test STP for bit-vector formulas as we encountered serious problems when we tried to use CVC3 to convert SMT formulas to CVC format which is needed by STP. However, we could test STP for a more restricted bit-vector logic combined with arrays. These results are shown in Tab. 4.2.

We detected incorrect results, i.e. solvers report *sat* although the status is *unsat* or vice versa, in the following way. First, we compared the result of each solver on each formula to the result of Boolector 0.4 and Z3 from SMT-COMP'08. If Boolector and Z3 agreed on the result, but the tested solver reported the opposite, we collected this formula.

Then, we inspected the collected formulas. Beaver claims that one formula is *sat* although other solvers report *unsat*. However, Beaver crashes with

an assertion failure when asked to provide a model. For CVC3 we could use its built-in on-the-fly proof checker to confirm nearly all cases where CVC3 wrongly concludes *unsat*. Moreover, most of the remaining wrong answers are caused by CVC3's query preprocessor. After disabling preprocessing, CVC3 reports the expected answer in most of the cases. For MathSAT we used the command line argument `-smtcomp` as it is documented in the MathSAT 4 invocation guide on their web page. However, we found out that this argument is responsible for many incorrect results where MathSAT spuriously reports *unsat*. If we use `-input=smt`, `tsolver=bv` and `-solve` instead of `-smtcomp`, MathSAT reports the expected results in almost every case. Similarly, Spear reports the expected results when common sub-expression elimination is turned off via `--cse 0`. The remaining incorrect results were confirmed by the majority voting principle where we used at least Boolector 1.1, Z3 from SMT-COMP'08 and another solver where error prone optimizations were turned off.

For the quantifier-free theory of bit-vectors, arrays and uninterpreted functions `QF_AUFBV`, we tested the following solvers: Boolector [BB09a] 1.0 and 1.1, a development version of CVC3 [BT07] 1.5 (downloaded April 29th, 2009), STP [GD07] 0.1 (November 18th, 2008), and Z3 [dMB08] 1.2 and from SMT-COMP'08 [BDOS08]. The results are summarized in Tab. 4.2.

Finally, we delta-debugged failure-inducing formulas found for `QF_BV`. Before delta-debugging, we semi-automatically divided them into bug classes with a limit of 50 formulas for each class. The results are shown in Tab. 4.3 and Tab. 4.4. We encountered only a few non-deterministic bugs that we were not able to delta-debug as they were not always reproducible. Incorrect results were delta-debugged as follows. Instead of calling the incorrect solver directly, the delta-debugger calls a shell script during delta-debugging. The script calls three trusted solvers and the incorrect solver. It returns 1 only if the three trusted solvers agree on the satisfiability status and the incorrect solver reports the opposite, and 0 otherwise.

Note that we did not delta-debug failure-inducing formulas found for `QF_AUFBV` as the inputs found for STP seemed to trigger non-deterministic behavior, and delta-debugging crashes for CVC3 would mainly consider de-

	no-div		guard-div	
solver	crash	incorrect	crash	incorrect
Beaver 1.1 rc1	0	0	12430	1
Boolector 1.0	0	0	0	0
Boolector 1.1	0	0	0	0
CVC3 1.5	902	8	-	-
MathSAT 4.2.3	0	113	2097	83
OpenSMT	19871	8	-	-
Spear 2.7	0	6	3577	71
Sword smt-comp	0	1	0	0
Z3 1.2	0	0	2264	0
Z3 smt-comp	0	0	0	0

Table 4.1: Experimental results of fuzzing bit-vector solvers. The file size of random bit-vector SMT formulas typically ranges from a few KB to 1 MB. The results are divided into bit-vector formulas without division operators (`no-div`) and formulas with "guarded" division (`guard-div`). Moreover, the results show the number of crashes, i.e. solver terminates in an unexpected way without providing a result, and number of incorrect results, i.e. solver reports *unsat* although formula is *sat*, or vice versa. Guarded division adds top-level constraints that rule out models where division by zero occurs. This guarantees that the semantics of dividing by zero does not influence the satisfiability status of the formula. We used a maximum bit-width of 16 for `no-div` formulas and tested each solver with the same set of 23100 randomly generated SMT formulas in three hours. For `guard-div` formulas that may additionally contain `bvudiv` and `bvurem`, we used a maximum bit-width of 10, as bit-vector division significantly slowed down some solvers. In this category we tested 23100 formulas in about one hour.

fects in bit-vector parts, which is what we already consider in our delta-debugging experiments for `QF_BV` in Tab. 4.3 and Tab. 4.4. However, in order to evaluate the effectiveness of our approach on formulas that contain arrays, we provide delta-debugging experiments for `QF_AX` formulas in Tab. 4.8.

4.4.2 Non-Bit-Vector Theories

We provide additional experimental results in order to show that our testing and debugging approaches are not limited to bit-vector theories. In partic-

solver	crash
CVC3 1.5	9812
STP 0.1	24

Table 4.2: Experimental results of fuzzing bit-vector and array solvers. The formulas contain bit-vector arrays, reads and writes, but no equalities between arrays as STP does not support them. We tested each solver with the same set of 12000 randomly generated formulas in about two hours. In order to be able to use CVC3 as filter to test STP, we had to restrict the set of operators which was used for the results in Tab. 4.1 even further, i.e. the formulas neither contain division nor shift operators. For Boolector and Z3 we could not find any defects. Generally, no incorrect results were found, but serious internal crashes. An analysis of the error messages showed that some crashes were caused by defects in the implementation of decision procedures.

	no-div				
solver	f	c	t	s	r
CVC3 1.5	139	9	172	2429	98%
MathSAT 4.2.3	50	1	10	611	97%
OpenSMT	154	4	5	492	96%
Spear 2.7	6	1	5	401	96%
Sword smt-comp	1	1	4	135	99%

Table 4.3: First experimental results of delta-debugging bit-vector solvers. The SMT formulas do not contain any division operators. The columns labelled f and c represent the number of formulas, and the number of bug classes. The number of classes is a rough approximation for the number of different solver defects. The columns t, s, and r show the average delta-debugging time in seconds, the average file size in bytes after delta-debugging, and the average file size reduction. We encountered some statistical outliers. The median of time t is 2 seconds for OpenSMT and 14 seconds for CVC3. The median of file size after delta-debugging s is 918 bytes for CVC3.

	guard-div				
solver	f	c	t	s	r
Beaver 1.1 rc1	469	12	5	319	98%
MathSAT 4.2.3	190	5	58	3709	76%
Spear 2.7	100	2	4	228	99%
Z3 1.2	50	1	734	254	99%

Table 4.4: Second experimental results of delta-debugging bit-vector solvers. The SMT formulas contain division operators that are guarded. The columns labelled f and c represent the number of formulas, and the number of bug classes. The number of classes is a rough approximation for the number of different solver defects. The columns t, s, and r show the average delta-debugging time in seconds, the average file size in bytes after delta-debugging, and the average file size reduction. We encountered some statistical outliers. The median of time t is 648 seconds for Z3. The medians for time t, file size s after delta-debugging, and reduction in file size r are 13 seconds, 715 bytes and 91% for MathSAT.

ular, we provide experimental results for the quantifier-free theory of arrays with indices and elements of uninterpreted sorts (`QF_AX`), and the quantifier-free theory of integer difference logic (`QF_IDL`). The failure-inducing inputs and delta-debugged results are publicly available[4].

We developed a new fuzzing tool called `FuzzSMT` in order to support non-bit-vector theories as well. It adapts the main ideas of our previous fuzzing tool `FuzzSMTBV` to non-bit-vector theories which is straight-forward. `FuzzSMT` can generate random formulas in the following SMT-LIB theories [BRST08]: `QF_A`, `QF_AUFBV`, `QF_AUFLIA`, `QF_AX`, `QF_BV`, `QF_IDL`, `QF_LIA`, `QF_LRA`, `QF_NIA`, `QF_NRA`, `QF_RDL`, `QF_UF`, `QF_UFBV`, `QF_UFIDL`, `QF_UFLIA`, `QF_UFLRA`,`QF_UFNIA`, `QF_UFNRA`, `QF_UFRDL`, `AUFLIA`, `AUFLIRA` and `AUFNIRA`.

In contrast to our previous fuzzer `FuzzSMTBV`, our new fuzzer `FuzzSMT` can handle almost all first-order theories that are supported by the SMT-LIB [BRST08] at the moment. It can handle natural and rational numbers, uninterpreted functions, predicates and sorts, and quantifiers. Supported theories include arrays with indices and elements of uninterpreted sorts,

[4] `www.fmv.jku.at/brummayer/fuzz-dd-smt2.tar.7z`

with or without extensionality (QF_A, QF_AX), difference logic with or without uninterpreted functions (QF_IDL, QF_UFIDL, QF_RDL, QF_UFRDL), linear arithmetic with or without uninterpreted functions (and arrays) (QF_LIA, QF_LRA, QF_UFLIA, QF_UFLRA, QF_AUFLIA), non-linear arithmetic with or without uninterpreted functions (QF_NIA, QF_NRA, QF_UFNIA, QF_UFNRA), and complex first-order theories such as AUFLIRA and AUFNIRA which, in essence, contain quantifiers, linear or non-linear arithmetic, and arrays of arrays with integer indices and real values.

For our experiments we used the same hardware and settings as for the previous bit-vector experiments. We ran our experiments under Ubuntu Linux on an Intel Core 2 Quad machine with 2.66 GHz and 8 GB RAM. Again, our fuzz testing framework used each of the four cores for testing. Our delta-debugging experiments were performed on the same machine, but were not run simultaneously.

For the quantifier-free theory of integer difference logic QF_IDL we used our fuzzer FuzzSMT in order to test the following solvers: Barcelogic SMT solver [BNO+08b] from SMT-COMP'08 [BDOS08], a development version of CVC3 [BT07] 1.5 (downloaded June 24th, 2009), MathSAT [BCF+07] 4.2.5, Sateen [KJS08] from SMT-COMP'08 [BDOS08], Yices [DdM06] 1.0.21 and Z3 [dMB08] from SMT-COMP'08 [BDOS08]. The results are shown in Tab. 4.5. As in the case of bit-vector theories, we found an impressive number of errors and incorrect results. This confirms that our fuzzing techniques are not limited to bit-vector theories.

Incorrect results were fully automatically classified by the majority voting principle with a minimum voting majority of 66 percents, i.e. more than 66 percents of the solvers that report a result have to agree on the satisfiability status in order to classify solvers reporting the opposite as incorrect. Instances on which less than 66 percents of the majority agree are classified as unresolved. In 36 out of 267 cases the voting majority was 66 percents. In the remaining 231 cases the voting majority was 83 percents. Unresolved discrepancies, i.e. the solvers disagree on the satisfiability status but the voting majority is less than 66 percents, did not occur. Note that the majority voting principle does not guarantee that results classified as incorrect are indeed

solver	error	incorrect
Barcelogic smt-comp	20	0
CVC3 1.5	0	0
MathSAT 4.2.5	72	19
Sateen smt-comp	190	229
Yices 1.0.21	0	0
Z3 smt-comp	0	19

Table 4.5: Experimental results of fuzzing integer difference logic solvers. The results show the number of errors, i.e. solver terminates in an unexpected way without providing a result, or it does not seem to terminate. Moreover, we show the number of incorrect results, i.e. solver reports *unsat* although formula is *sat*, or vice versa. We tested each solver with the same set of 24070 randomly generated SMT formulas. Finally, the overall testing time was about 35 minutes.

incorrect. From a theoretical point of view, they could also be correct although the majority of the other solvers report the opposite result. However, our fuzzing experience suggests that this case is unlikely to happen.

For each SMT solver call we used a time limit of 60 seconds. Only Barcelogic exceeded the time limit in some cases. We inspected some samples and found out that although the other solvers immediately terminate on these small benchmarks, Barcelogic does not terminate, even after hours. Therefore, we classified these cases as errors.

The errors reported for Sateen are mainly segmentation faults and aborts. In two cases Sateen printed out "UNSAT" followed by a line reporting "unknown". In one case Sateen printed out "UNSAT", however, followed by a line reporting "sat". Moreover, Sateen sometimes showed non-deterministic behavior and reported *unknown* in 372 cases. We did not classify these cases as errors although the formulas are decidable. Finally, the errors reported for MathSAT consist of segmentation faults.

For the extensional quantifier-free theory of arrays with indices and elements of uninterpreted sorts `QF_AX`, we used our fuzzer `FuzzSMT` in order to test the following solvers: Barcelogic SMT solver [BNO+08b] from SMT-COMP'08 [BDOS08], a development version of CVC3 [BT07] 1.5 (down-

solver	error	incorrect
Barcelogic smt-comp	25	54
CVC3 1.5	0	0
Z3 smt-comp	0	0

Table 4.6: Experimental results of fuzzing solvers for arrays with indices and elements of uninterpreted sorts. The results show the number of errors, i.e. solver terminates in an unexpected way without providing a result, or it does not seem to terminate. Moreover, we show the number of incorrect results, i.e. solver reports *unsat* although formula is *sat*, or vice versa. We tested each solver with the same set of 24030 randomly generated SMT formulas. The overall testing time was about 40 minutes.

loaded June 24th, 2009) and Z3 [dMB08] from SMT-COMP'08 [BDOS08]. The results are shown in Tab. 4.6.

We used a timeout of 60 seconds. As expected, we found errors and incorrect results. Again, incorrect results were fully automatically classified by the majority voting principle with a minimum voting majority of 66 percents. Unresolved discrepancies, e.g. two solvers disagree while the third solvers crashes, did not occur.

Analogously to the experimental results for `QF_IDL`, we found some failure-inducing formulas on which Barcelogic does not seem to terminate. However, we found other errors as well. On three instances Barcelogic terminates immediately without printing out a result. Moreover, we found two segmentation faults and two assertion failures.

We extended our delta-debugger `DeltaSMT`. As our delta-debugging algorithm for SMT formulas, presented in section 4.3, is generic in the sense that it does not depend on a specific theory, we did not have to change it. The main work was to extend the parser and the type universe. The results for `QF_IDL` are shown in Tab. 4.7 and the results for `QF_AX` are shown in Tab. 4.8. Our experimental results clearly confirm the effectiveness of our delta-debugging approach on non-bit-vector theories.

Delta-debugging Barcelogic was problematic as there are formula structures on which Barcelogic does not seem to terminate. This causes the following two problems. First, we must find a new strategy in order to delta-debug

solver	f	c	t	s	r
Barcelogic smt-comp	20	1	29	979	55%
MathSAT 4.2.5	69	2	1	191	92%
Sateen smt-comp	103	4	3	1231	46%
Z3 smt-comp	17	1	3	614	71%

Table 4.7: Experimental results of delta-debugging integer difference logic solvers. The columns labelled f and c represent the number of formulas, and the number of bug classes. The columns t, s, and r show the average delta-debugging time in seconds, the average file size in bytes after delta-debugging, and the average file size reduction. Note that due to non-deterministic behavior a few failures were not always reproducible and could therefore not be delta-debugged. We had the following statistical outliers. The median of reduction in file size r is 73% for Barcelogic and 92% for Z3. Moreover, the median of file size after delta-debugging s is 619 bytes for Barcelogic, 176 bytes for MathSAT, 879 bytes for Sateen and 174 bytes for Z3.

failure-inducing formulas on which a solver does seem to terminate. Second, if we perform delta-debugging with respect to other failure-inducing formulas, i.e. formulas that lead to a segmentation fault, it may happen that the result of a simplification step is a formula on which the solver does not terminate. Thus, the overall delta-debugging algorithm may not terminate anymore. We solved both problems with the concept of timeouts.

In order to solve the first problem, we wrote a shell script that executes Barcelogic with a time limit. If Barcelogic exceeds the time limit we return 1, and 0 otherwise. This script is called by the delta-debugger after each simplification step in order to find out whether the simplified formula still triggers an infinite loop within the solver or not. In this way, we delta-debugged the 20 `QF_IDL` instances shown in Tab. 4.7. We found out that a time limit of 2 seconds is enough as the formulas and its simplifications were rather easy to solve, i.e. a timeout of 2 seconds is very likely to be adequate.

In order to solve the second problem, we used the timeout feature of our delta-debugger `DeltaSMT`. The delta-debugger can be configured to use a time limit for each simplification call. If the solver exceeds this limit, the delta-debugger treats this case as if the current simplifications has failed,

solver	f	c	t	s	r
Barcelogic smt-comp	75	6	73	3286	81%

Table 4.8: Experimental results of delta-debugging Barcelogic for `QF_AX` formulas. The columns labelled f and c represent the number of formulas, and the number of bug classes. The columns t, s, and r show the average delta-debugging time in seconds, the average file size in bytes after delta-debugging, and the average file size reduction. We encountered the following statistical outliers. The median of time t is 30 seconds and the median of reduction in file size r is 91%. Moreover, the median of file size after delta-debugging s is 1515 bytes.

i.e. the current simplified formula does not trigger the bug anymore, and backtracks. In this way we were able to delta-debug failure-inducing inputs that trigger assertion failures, segmentation faults and immediate terminations without providing a result, although the solver might not terminate after each simplification step. For example, assume we delta-debug a defect such as a segmentation fault. Whenever the delta-debugger simplified the current formula in such a way that Barcelogic does not terminate, then Barcelogic exceeds the time limit, the delta-debugger treats this case as a simplification on which the original kind of defect,i.e. a segmentation fault, does not occur anymore, and backtracks. With these two concepts of timeouts we were able to delta-debug all kind of defects found for Barcelogic in `QF_AX`. The results are shown in `tableddax`.

4.5 Conclusion

The goal of our experiments is to show that traditional testing techniques obviously do not suffice to fulfill the high demand for robustness and correctness of SMT solvers. SMT solvers contain a lot of error prone optimizations that need to be heavily tested. Although fuzz testing is not a complete solution for this problem, i.e. it can only find bugs but cannot prove the absence, we have shown that it is a useful "tool in the toolbox" which can find many bugs in state-of-the-art SMT solvers that traditional testing techniques ob-

viously miss. Our first experimental evaluations showed the effectiveness of our approaches for bit-vector theories. Our additional experiments for non-bit-vector theories confirm that our fuzzing and delta-debugging techniques are *not* limited to bit-vector theories and, moreover, can be used to debug solvers that may not terminate on every input.

We conclude that fuzz testing in combination with delta-debugging is an effective approach which can be easily integrated in the development process of SMT solvers in order to increase robustness and correctness. Moreover, we believe that this combination can be of great value in other domains as well. Blackbox fuzzing is able to find many defects, even without any knowledge of implementation details. It is therefore a reasonable conjecture that using knowledge of these details in a whitebox fuzzing approach may be even more successful in finding more, or more subtle, defects.

Chapter 5

Conclusion

The problem statement of this thesis is to design, implement, test, and debug an efficient SMT solver for the extensional theory of arrays, combined with bit-vectors. Each chapter of this thesis addressed different aspects of this problem. Detailed experimental evaluations confirmed the success of our main approaches.

Chapter two addressed the theoretical development of a novel and efficient decision procedure for the extensional quantifier-free theory of arrays. We discussed the decision procedure from a theoretical point of view, i.e. we provided a complexity analysis and proofs, but also discussed implementation and optimization details. Experimental evaluations clearly showed the effectiveness of our novel decision procedure.

Chapter three focused on the architectural design of an efficient SMT solver, called Boolector, and discussed selected optimization techniques and features such as symbolic overflow detection, propagating unconstrained variables, and under-approximation techniques for bit-vectors and reads on bit-vector arrays. An experimental analysis of our under-approximation techniques showed a speed-up for Boolector on satisfiable instances, which is in particular promising if Boolector is run on instances that are expected to be satisfiable. Moreover, the success of Boolector at the SMT competitions [BDOS08, BDOS09] in 2008 and 2009 confirms the overall success of our techniques. In the SMT competitions 2008 and 2009, Boolector clearly won

the division of the quantifier-free theory of bit-vectors, arrays and uninterpreted functions `QF_AUFBV`. Moreover, it won the division of the quantifier-free theory of bit-vectors `QF_BV` in 2008 and achieved the second place in 2009.

In chapter four we addressed the neglected problem of testing and debugging SMT solvers. In particular, we showed that most state-of-the-art SMT solvers were broken at the time of our tests. They contained defects that lead to crashes and incorrect results. An experimental analysis clearly showed that our fuzzing techniques were able to find many bugs that traditional testing techniques were obviously not able to find. Moreover, we showed that our generic delta-debugging techniques were able to efficiently minimize failure-inducing formulas, even in cases where SMT solvers do not seem to terminate.

In contrast to other scientific publications on SMT, where theoretical aspects are presented solely, this thesis provides implementation and optimization details. Confirmed by our experimental results of testing SMT solvers, we think that more papers should be published on practical aspects of SMT solver development such as testing, debugging and providing implementation details. Theoretical decision procedures, although interesting and beautiful from a theoretical point of view, are hardly useful if we are not able to implement them *efficiently* and *correctly*. More research has to be done in order to attack these problems.

5.1 What's next?

There are various directions that seem to be promising for the SMT community. First of all, as nowadays desktop computers have multiple cores, it sounds straight-forward to develop concurrent solvers. However, the issue is not as straight-forward as one might expect. At the moment, the most successful approach is a portfolio-based approach as implemented in ManySAT [HJS09]. Multiple SAT solvers with different algorithms and heuristics are run in parallel. To improve performance, learned clauses are propagated. In [WHdM09], Wintersteiger et al. adapted this approach to SMT. However, it is still unclear if this portfolio-based approach is indeed

the best approach. More research is needed in order to understand parallel SAT and SMT solving techniques. In particular, more research has to be done on analyzing the trade-off between parallel solving and cache performance, which is a critical aspect in overall solving time. Moreover, we suggest that also non-DPLL based approaches should be considered. The predominant focus on DPLL-based approaches may lead to the effect that other efficient parallel solving techniques are not discovered. Nevertheless, as the number of cores is steadily increasing, parallel SAT and SMT seem to be interesting and promising approaches.

Craig interpolants are a promising research direction for SMT. McMillan introduced propositional interpolants [McM03] in order to perform *unbounded* symbolic model checking. Recently, research focuses on computing theory-specific interpolants [FGG+09, CGS09, GKT09, KV09, CGS08, KW07, KMZ06, YM05, McM04] in order to enable and improve formal verification techniques. Interesting aspects of future research on interpolants are the support of further theories, efficient interpolation generation algorithms, and interpolant minimization.

Due to the limitation of the Nelson-Oppen Framework [NO79], SMT solvers typically support only a limited form of quantifier reasoning. Recent research on quantifier handling [GdM09, GBT07] tries to close this gap. Efficient quantifier reasoning has great potential as it enables more complex verification tasks. Interesting aspects of future research are the support of more complex theories that support quantifiers, but are still decidable, and to improve the overall performance of quantifier reasoning.

Finally, there is still great potential in developing and optimizing efficient decision procedures for various theories of interest. We hope that we can find more powerful decision procedures and optimization techniques in the future. It will be exciting to see the (hopefully) never-ending progress of the SMT community in the next years.

Bibliography

[Ack54] W. Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1954.

[Art08] C. Artho. Iterative Delta Debugging. In *Proceedings of the* 4^{th} *Haifa Verification Conference (HVC)*. Springer-Verlag, 2008.

[Bab07] D. Babić. SPEAR Modular Arithmetic Format Specification - Version 1.0 - Revision 1.7, December 21^{th} 2007. `www.cs.ubc.ca/~babic/doc/spear_modarith.pdf`.

[Bab08] D. Babić. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[Bar03] C. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, 2003.

[BB04] C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV)*, pages 515–518. Springer-Verlag, 2004.

[BB06] R. Brummayer and A. Biere. Local Two-Level And-Inverter Graph Minimization without Blowup. In *Proc. MEMICS'06*, pages 32–38. Faculty of Information Technology, Brno University, 2006.

[BB07] R. Brummayer and A. Biere. C32SAT: Checking C Expressions. In *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV)*, pages 294–297. Springer-Verlag, 2007.

[BB08a] A. Biere and R. Brummayer. Consistency Checking of All Different Constraints over Bit-Vectors within a SAT-Solver. In *Proceedings of the* 8^{th} *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 223–226. IEEE, 2008.

[BB08b] R. Brummayer and A. Biere. Lemmas on Demand for the Extensional Theory of Arrays. In *Proceedings of the joint Workshops of the* 6^{th} *International Workshop on Satisfiability Modulo Theories and the* 1^{st} *International Workshop on Bit-Precise Reasoning*, pages 6–11. ACM, 2008.

[BB09a] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proceedings of the* 15^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 174–177. Springer-Verlag, 2009.

[BB09b] R. Brummayer and A. Biere. Effective Bit-Width and Under-Approximation. In *Proceedings of the* 12^{th} *International Conference on Computer Aided Systems Theory*. Springer-Verlag, 2009.

[BB09c] R. Brummayer and A. Biere. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the* 7^{th} *International Workshop on Satisfiability Modulo Theories*, 2009.

[BB09d] R. Brummayer and A. Biere. Lemmas on Demand for the Extensional Theory of Arrays. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6:165–201, 2009.

[BBC+05] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In *Proceedings of the* 17^{th} *International Conference on Computer Aided Verification (CAV)*, pages 335–349. Springer-Verlag, 2005.

[BBC+06] M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzen, Z. Hanna, Z. Khasidashvili, A. Palti, and R. Sebastiani. Encoding RTL Constructs for MathSAT: a Preliminary Report. In *Proceedings of the* 3^{rd} *International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*. Elsevier, 2006.

[BBL08] R. Brummayer, A. Biere, and F. Lonsing. BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking. In *Proceedings of the joint Workshops of the* 6^{th} *International Workshop on Satisfiability Modulo Theories and the* 1^{st} *International Workshop on Bit-Precise Reasoning*, pages 33–38. ACM, 2008.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the* 5^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 193–207. Springer-Verlag, 1999.

[BCF+06a] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, A. Santuari, and R. Sebastiani. To Ackermann-ize or Not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in SMT ($\mathcal{EUF} \cup \mathcal{T}$). In *Proceedings of the* 13^{th} *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 557–571. Springer-Verlag, 2006.

[BCF+06b] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for

Satisfiability Modulo Theories: A Comparative Analysis. In *Proceedings of the* 13^{th} *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 527–541. Springer-Verlag, 2006.

[BCF+07] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A Lazy and Layered SMT($\mathcal{BV}$) Solver for Hard Industrial Verification Problems. In *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV)*, pages 547–560. Springer-Verlag, 2007.

[BDOS08] C. Barrett, M. Deters, A. Oliveras, and A. Stump. SMT-Comp, 2008. `www.smtcomp.org/2008`.

[BDOS09] C. Barrett, M. Deters, A. Oliveras, and A. Stump. SMT-Comp, 2009. `www.smtcomp.org/2009`.

[BDS02] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV)*, pages 236–249. Springer-Verlag, 2002.

[BFG+05] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A Translation Validator for Optimizing Compilers. In *Proceedings of the* 17^{th} *International Conference on Computer Aided Verification (CAV)*, pages 291–295. Springer-Verlag, 2005.

[BHvMW09] A. Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability.* IOS Press, 2009.

[Bie08] A. Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.

[Bie09] A. Biere. The AIGER And-Inverter Graph (AIG) Format, 2009. `fmv.jku.at/aiger`.

[BKO+07] R. Bryant., D. Kroening, J. Ouaknine, S. Seshia, O. Strichman, and B. Brady. Deciding Bit-Vector Arithmetic with Abstraction. In *Proceedings of the* 13^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 358–372. Springer, 2007.

[BKO+09] R. Bryant, D. Kroening, J. Ouaknine, S. Seshia, O. Strichman, and B. Brady. Deciding Bit-Vector Arithmetic with Abstraction. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(2):95–104, 2009.

[BM07] A. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag, 2007.

[BMS06] A. Bradley, Z. Manna, and H. Sipma. What's Decidable About Arrays? In *Proceedings of the* 7^{th} *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 427–442. Springer-Verlag, 2006.

[BN04] S. Berghofer and T. Nipkow. Random Testing in Isabelle/HOL. In *Proceedings of the* 2^{nd} *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 230–239. IEEE, 2004.

[BNO+08a] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodriguez-Carbonell, and A. Rubio. A Write-Based Solver for SAT Modulo the Theory of Arrays. In *Proceedings of the* 8^{th} *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2008.

[BNO+08b] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. The Barcelogic SMT Solver. In

Proceedings of the 20^{th} *International Conference on Computer Aided Verification (CAV)*, pages 294–298, 2008.

[BNOT06] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on Demand in SAT Modulo Theories. In *Proceedings of the* 15^{th} *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 512–526. Springer-Verlag, 2006.

[BRST08] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[Bru08] Roberto Bruttomesso. *RTL Verification: From SAT to SMT(BV)*. PhD thesis, University of Trento, 2008.

[BS08] R. Bruttomesso and N. Sharygina. OpenSMT 0.1 System Description, 2008.

[BT07] C. Barrett and C. Tinelli. CVC3. In *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV)*, pages 298–302. Springer-Verlag, 2007.

[BTV09] N. Bjørner, N. Tillmann, and A. Voronkov. Path Feasibility Analysis for String-Manipulating Programs. In *Proceedings of the* 15^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 307–321. Springer-Verlag, 2009.

[CDE08] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the* 8^{th} *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 209–224. USENIX Association, 2008.

[CGJ+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic

Model Checking. *Journal of the ACM (JACM)*, pages 752–794, 2003.

[CGP+06] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the* 13^{th} *ACM Conference on Computer and Communications Security*, pages 322–335. ACM, 2006.

[CGS08] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In *Proceedings of the* 14^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 397–412. Springer-Verlag, 2008.

[CGS09] A. Cimatti, A. Griggio, and R. Sebastiani. Interpolant Generation for UTVPI. In *Proceedings of the* 22^{nd} *International Conference on Automated Deduction (CADE)*, pages 167 – 182. Springer-Verlag, 2009.

[CH00] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the* 5^{th} *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000.

[CKL04] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs . In *Proceedings of the* 10^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176. Springer-Verlag, 2004.

[Cla08] E. Clarke. The Birth of Model Checking. In *25 Years of Model Checking*, pages 1–26. Springer-Verlag, 2008.

[Coo71] S. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the* 3^{rd} *ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[CS03] K. Claessen and N. Sörensson. New Techniques that Improve MACE-style Finite Model Finding. In *CADE-19, Workshop W4, Model Computation – Principles, Algorithms, Applications*, 2003.

[DdM06] B. Dutertre and L. de Moura. The Yices SMT Solver, 2006. `http://yices.csl.sri.com/tool-paper.pdf`.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5, 1962.

[dMB08] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the* 14^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer-Verlag, 2008.

[dMR02] L. de Moura and H. Rueß. Lemmas on Demand for Satisfiability Solvers. In *Proceedings of the* 5^{th} *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 244–251. Springer-Verlag, 2002.

[DNS05] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM (JACM)*, 52:365–473, 2005.

[Eén05] N. Eén. *SAT Based Model Checking*. PhD thesis, Chalmers University of Technology and Göteborg University, 2005.

[ES03] N. Eén and N. Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 89(4), 2003.

[ES04] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *Proceedings of the* 7^{th} *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 333–336. Springer-Verlag, 2004.

[FGG+09] A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli. Ground Interpolation for the Theory of Equality. In *Proceedings of the* 15^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 413–427. Springer-Verlag, 2009.

[FJS03] C. Flanagan, R. Joshi, and J. Saxe. Theorem Proving Using Lazy Proof Explication. In *Proceedings of the* 15^{th} *International Conference on Computer Aided Verification (CAV)*, pages 355–367. Springer-Verlag, 2003.

[Gan07] V. Ganesh. *Decision Procedures for Bit-Vectors, Arrays and Integers*. PhD thesis, Computer Science Department, Stanford University, 2007.

[GBT07] Y. Ge, C. Barrett, and C. Tinelli. Solving Quantified Verification Conditions using Satisfiability Modul Theories. In *Proceedings of the* 21^{st} *International Conference on Automated Deduction (CADE)*, pages 167–182. Sringer-Verlag, 2007.

[GD07] V. Ganesh and D. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV)*, pages 519–531. Springer-Verlag, 2007.

[GdM09] Y. Ge and L. de Moura. Complete instantiation for quantified smt formulas. In *Proceedings of the* 21^{st} *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2009.

[GK05] A. Groce and D. Kroening. Making the Most of BMC Counterexamples . In *Proceedings of the* 2^{nd} *International Workshop on Bounded Model Checking (BMC)*, pages 67–81. Elsevier, 2005.

[GKF08] A. Goel, S. Krstic, and A. Fuchs. Deciding Array Formulas with Frugal Axiom Instantiation. In *Proceedings of the joint*

Workshops of the 6^{th} *International Workshop on Satisfiability Modulo Theories and the* 1^{st} *International Workshop on Bit-Precise Reasoning*, pages 12–17. ACM, 2008.

[GKL04] A. Groce, D. Kroening, and F. Lerda. Understanding Counterexamples with explain . In *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV)*, pages 453–456. Springer-Verlag, 2004.

[GKL08] P. Godefroid, A. Kiezun, and M. Levin. Grammar-Based Whitebox Fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, pages 206–215. ACM, 2008.

[GKT09] A. Goel, S. Krstic, and C. Tinelli. Ground Interpolation for Combined Theories. In *Proceedings of the* 22^{nd} *International Conference on Automated Deduction (CADE)*, pages 183 – 198. Springer-Verlag, 2009.

[GLM08] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the* 16^{th} *Symposium on Network and Distributed System Security (NDSS)*. The Internet Society, 2008.

[GNRZ07] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision Procedures for Extensions of the Theory of Arrays. *Annals of Mathematics and Artificial Intelligence*, 50:231–254, 2007.

[GSK98] C. Gomes, B. Selman, and H. Kautz. Boosting Combinatorial Search Through Randomization. In *Proceedings of the* 15^{th} *International Conference on Artificial Intelligence (AAAI)*, pages 431–437. AAAI Press, 1998.

[HBG04] Y. Hu, C. Barrett, and B. Goldberg. Theory and Algorithms for the Generation and Validation of Speculative Loop Optimizations. In *Proceedings of the* 2^{nd} *IEEE International Conference*

on Software Engineering and Formal Methods (SEFM), pages 281–289. IEEE, 2004.

[HH07] N. He and M. Hsiao. Bounded Model Checking of Embedded Software in Wireless Cognitive Radio Systems. In *Proceedings of the* 25^{th} *International Conference on Computer Design (ICCD)*, pages 19–24. IEEE, 2007.

[HH08] N. He and M. Hsiao. A new Testability Guided Abstraction to Solving Bit-Vector Formula. In *Proceedings of the joint Workshops of the* 6^{th} *International Workshop on Satisfiability Modulo Theories and the* 1^{st} *International Workshop on Bit-Precise Reasoning*, pages 39–45. ACM, 2008.

[HHK08] T. Henzinger, T. Hottelier, and Laura Kovacs. Valigator: A Verification Tool with Bound and Invariant Generation. In *Proceedings of the* 15^{th} *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 333–342. Springer-Verlag, 2008.

[HJS09] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6:245–262, 2009.

[ISO99] ISO/IEC. Programming Languages - C (ISO/IEC 9899:1999(E)), 1999.

[JES07] P. Jackson, B. Ellis, and K. Sharp. Using SMT Solvers to Verify High-Integrity Programs. In *Proceedings of the* 2^{nd} *workshop on Automated Formal Methods (AFM)*, pages 60–68. ACM, 2007.

[JLS09] S. Jha, R. Limaye, and S. Seshia. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In *Proceedings of the* 21^{st} *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2009.

[KGP01] A. Kühlmann, M. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. In *Proceedings of the* 38^{th} *Conference on Design Automation (DAC)*, pages 232–237. ACM, 2001.

[KJS08] H. Kim, H. Jin, and F. Somenzi. Sateen: Sat Enumeration Engine for SMT-COMP'08, 2008. `www.smtexec.org/exec/competitors2008.php?desc=390`.

[KMZ06] D. Kapur, R. Majumdar, and C. Zarba. Interpolation for Data Structures. In *Proceedings of the* 14^{th} *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE))*, pages 105–116. ACM, 2006.

[KS07] D. Kroening and S. Seshia. Formal Verification at Higher Levels of Abstraction . In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 572–578. IEEE, 2007. Tutorial.

[KS08] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer-Verlag, 2008.

[KV09] L. Kovacs and A. Voronkov. Interpolation and Symbol Elimination. In *Proceedings of the* 22^{nd} *International Conference on Automated Deduction (CADE)*, pages 199 – 213. Springer-Verlag, 2009.

[KW07] D. Kroening and G. Weissenbacher. Lifting Propositional Interpolants to the Word-Level . In *Proceedings of the* 7^{th} *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 85–89. IEEE, 2007.

[LQ08] S. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. In *Proceedings of the* 35^{th} *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 171–182. ACM, 2008.

[LS04] S. Lahiri and S. Seshia. The UCLID Decision Procedure. In *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV)*, pages 475–478. Springer-Verlag, 2004.

[McC62] J. McCarthy. Towards a Mathematical Science of Computation. In *Proceedings of the IFIP Congress*, pages 21–28. North-Holland, 1962.

[McC94] W. McCune. A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical report, Argonne National Laboratory, 1994.

[McM92] K. McMillan. *Symbolic Model Checking. An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.

[McM03] K. McMillan. Interpolation and SAT-Based Model Checking. In *Proceedings of the* 15^{th} *International Conference on Computer Aided Verification (CAV).* Springer-Verlag, 2003.

[McM04] K. McMillan. An Interpolating Theorem Prover. In *Proceedings of the* 10^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 101–121. Springer-Verlag, 2004.

[MKL+95] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A.Natarajan, and J. Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report CS-TR-1995-1268, University of Wisconsin, Madison, 1995.

[MM00] P. Manolios and J. Moore. *Computer-Aided Reasoning: An Approach.* Kaufmann, 2000.

[MS06] G. Misherghi and Z. Su. HDD: Hierarchical Delta Debugging. In *Proceedings of the* 28^{th} *International Conference on Software Engineering (ICSE)*, pages 142–151. ACM, 2006.

[MSV06] P. Manolios, S. Srinivasan, and D. Vroon. Automatic Memory Reductions for RTL Model Verification. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 786–793. ACM, 2006.

[NO79] G. Nelson and D. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1:245–257, 1979.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM (JACM)*, 53:937–977, 2006.

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2002.

[OSR92] S. Owre, N. Shankar, and J. Rushby. PVS: A Prototype Verification System. In *Proceedings of the* 11^{th} *International Conference on Automated Deduction (CADE)*. Springer-Verlag, 1992.

[Owr06] S. Owre. Random Testing in PVS. In *Proceedings of the* 1^{st} *Workshop on Automated Formal Methods (AFM)*, 2006.

[PD07] K. Pipatsrisawat and A. Darwiche. RSat 2.0: SAT Solver Description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[RT06] S. Ranise and C. 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`.

[SBDL01] A. Stump, C. Barrett, D. Dill, and J. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *Proceedings of the* 16^{th} *Symposium on Logic in Computer Science (LICS)*, pages 29–37. IEEE, 2001.

[Seb07] R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 3:141–224, 2007.

[SGA07] M. Sutton, A. Greene, and P. Amini. *Fuzzing - Brute Force Vulnerability Discovery.* Pearson Education, 2007.

[SGBB00] M. Schulte, M. Gok, P. Balzola, and R. Brocato. Combined Unsigned and Two's Complement Saturating Multipliers, 2000.

[SGF09] S. Srivastava, S. Gulwani, and J. Foster. VS^3: SMT Solvers for Program Verification. In *Proceedings of the* 21^{st} *International Conference on Computer Aided Verification (CAV).* Springer-Verlag, 2009.

[Sho84] R. Shostak. Deciding Combinations of Theories. *Journal of the ACM (JACM)*, 31:1–12, 1984.

[Sin06] Eli Singerman. Challenges in Making Decision Procedures Applicable to Industry. In *Proceedings of the* 3^{rd} *International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR).* Elsevier, 2006.

[SJPS08] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An Automatic Verifier for Java-Like Programs Based on Dynamic Frames. In *Proceedings of the* 11^{th} *Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 261–275. Springer-Verlag, 2008.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Proceedings of the*

3^{rd} *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 108–125. Springer-Verlag, 2000.

[TdH08] N. Tillmann and J. de Halleux. PEX - White Box Test Generation for .NET. In *Proceedings of the* 2^{nd} *International Conference on Tests and Proofs (TAP)*, pages 134–153. Springer-Verlag, 2008.

[TDM08] A. Takanen, J. Demott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[Tse83] G. Tseitin. On the Complexity of Proofs in Propositional Logics. *Automation of Reasoning: Classical Papers in Computational Logic 1967-1970*, 2, 1983. Originally published 1970.

[Vid08] A. Vida. Random Test Case Generation and Delta Debugging for BitVector Logic with Arrays. Master's thesis, Johannes Kepler University, Linz, Austria, 2008.

[VTP09] D. Vanoverberghe, N. Tillmann, and F. Piessens. Test Input Generation for Programs with Pointers. In *Proceedings of the* 15^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 277–291. Springer-Verlag, 2009.

[WFG+07] R. Wille, G. Fey, D. Groe, S. Eggersgl, and R. Drechsler. SWORD: A SAT like Prover Using Word Level Information. In *Proceedings of the* 15^{th} *IFIP International Conference on Very Large Scale Integration (VLSI)*, pages 88–93. IEEE, 2007.

[WHdM09] C. Wintersteiger, Y. Hamadi, and L. de Moura. A Concurrent Portfolio Approach to SMT Solving. In *Proceedings of the* 21^{st} *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2009.

[Wir95] N. Wirth. *Digital Circuit Design: An Introductory Textbook*. Springer-Verlag, 1995.

[YM05] G. Yorsh and M. Musuvathi. A Combination Method for Generating Interpolants. In *Proceedings of the* 20^{th} *International Conference on Automated Deduction (CADE)*, pages 353–368. Springer-Verlag, 2005.

[Zel05] A. Zeller. *Why Programs Fail. A Guide to Systematic Debugging.* Kaufmann, 2005.

[ZH02] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28:183–200, 2002.

[ZPG+05] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and Run-Time Validation of Loop Transformations. *Formal Methods in System Design (FMSD)*, 27(3):335–360, 2005.

[ZZ96] H. Zhang and J. Zhang. Generating Models by SEM. In *Proceedings of the* 13^{th} *International Conference on Automated Deduction (CADE)*, pages 308 – 312. Springer-Verlag, 1996.

Appendix A

The BTOR Format

A.1 Overview

BTOR is a quantifier-free word-level format for formulas over bit-vectors in combination with one-dimensional arrays. It is strongly typed, easy to parse, multi-rooted, and has precise semantics. Unlike the Spear Format (SF) [Bab07], which only supports bit-vectors up to 64 bits, bit-vectors can have an arbitrary bit-width in BTOR.

BTOR supports bit-vector variables, constants, and one-dimensional bit-vector arrays. Most bit-vector operations can be used in a signed or unsigned context. In the signed context, bit-vectors are interpreted as being represented in two's complement.

Similar to SF, the result of every operation is assigned to an intermediate variable, as the following example shows:

```
1 var 32            3 constd 32 127          5 eq 1 3 4
2 var 32            4 and 32 1 -2            6 root 1 5
```

In line 1 and 2 we declare two 32-bit variables. In BTOR, a non-negative integer in the first column is used as unique identifier. Typically, we use the line number as identifier. In line 3 we declare the decimal 32-bit constant 127. Line 4 shows how computation is expressed in BTOR. We use the bit-wise operator *and* and apply it to the operands with the identifiers 1 and 2, which are the variables we have declared before. The result of this operation

has the identifier 4. The minus before 2 expresses logical negation[1], i.e. we flip all bits of the variable 2 before we combine it with the variable 1. The result has 32 bits, which is indicated by the bit-width after the operator. In line 5 we compare the decimal constant with the sub-formula 4 for equality. The length of this result is 1, as relational operators are boolean. Finally, we declare the sub-formula 5 as boolean root.

This simple example already shows some interesting design decisions. The format should be easy to parse. The first column is used for the identifier, the second for the operator, and the third for the bit-width. This restriction makes it easy to write a parser for BTOR. The additional type information in the third column makes it easy for the parser to check type consistency on the fly.

Being able to add back annotation in form of an explicit symbol table simplifies many applications. In BTOR there is no separate symbol table as in the And-Inverter Graph Format AIGER [Bie09]. However, variable declarations can have an additional symbol part, separated by white space after the bit width. Comments start with ';' and stretch until the end of the current line.

BTOR does not allow forward references, i.e. all operands have to be declared before they are used. This restriction makes parsing BTOR instances trivial, as it can be done in one pass. For example, our visualizer of the BTOR format consists of an AWK script of less than 80 lines, and produces a graph description in DOT format. A utility that prints a histogram of the number of operator occurrences can be implemented as the following one line AWK script:

```
{a[$2]++}END{for(k in a)printf "%-7s%d\n", k, a[k]|"sort -n -k 2";}
```

As in Verilog, boolean variables are treated just as bit-vectors with bit-width one. This simplifies the format as we do not have to convert from boolean to bit-vector and vice versa. We can simply treat the boolean case as bit-vector case with bit-width one.

BTOR is multi-rooted, which allows to model multiple outputs. Multiple

[1]Note that BTOR also supports the unary but redundant operator *not*.

roots can be used to model hardware systems, which typically have boolean and bit-vector outputs.

A.2 Bit-vectors

BTOR supports the following bit-vector constructors: *var* for bit-vector variables, *constd*, *consth* and *const* for decimal, hexadecimal and binary constants, and *one*, *ones* and *zero* for the constants 1, -1 and 0. All constructors take the bit-width as first argument. The constructor *var* takes an optional symbol string as second argument. The constructors *constd*, *consth* and *const* take the constant value as second argument.

The set of bit-vector operators is shown in table A.1, A.2, A.3 and A.4. The columns w_1 to w_3 represent the bit-width of the operands. The column w_r represents the bit-width of the result. The semantics of most operators are defined by the semantics of the corresponding operators in the quantifier-free theory of fix-sized bit-vectors `QF_BV` in the SMT-LIB standard [RT06]. The only exceptions are as follows.

The SMT-LIB standard does not specify the result of dividing by zero. In BTOR the result of dividing by zero is the largest unsigned integer that can be represented with the bit-width of the operands. This corresponds to real divider circuits in hardware systems. Nevertheless, the underspecified variant of the SMT-LIB format can always be modelled by treating divison by zero as uninterpreted function.

The bit-width of shift operands are restricted in the following way. The bit-width of the first operand has to be a power of two. The bit-width of the second argument has to be log_2 of the bit-width of the first operand. If the bit-width of the shift operands is log_2, then it is impossible to shift more than the bit-width, which for example is undefined in the programming language C [ISO99, BB07].

Additionally, BTOR supports the Verilog reduction operators *redand*, *redor* and *redxor*, the VHDL rotate operators *rol* and *ror*, and a new set of overflow detection operators. For example the overflow detection operator *umulo* returns 1 if unsigned multiplication overflows. Consider the following

class	operators	w_1	w_r
negation	**not**, neg	n	n
reduction	redand, redor, redxor	n	1
arithmetic	inc, dec	n	n

Table A.1: The unary bit-vector operators *not* and *neg* apply one's resp. two's complement. The operators *redand*, *redor* and *redxor* are reduction operators from Verilog. The operators *inc* and *dec* are used to increment resp. decrement a bit-vector by one.

example:

```
1 var 32
2 var 32
3 redand 1 1
4 redand 1 2
5 umulo 1 1 2
6 and 1 3 4
7 and 1 6 -5
8 root 1 7
```

The reduction operator *redand* returns 1, if all bits of the operand are set to one. In this case unsigned multiplication overflows. We assert the opposite in line 7, which makes this instance *unsatisfiable*.

Whether we want to detect arithmetic overflows or not, depends on the application scenario. Typically, in software verification we are interested in detecting overflows, as the results are often undefined and depend on compliler semantics [BB07]. However, in hardware verification overflow detection is in most cases unnecessary. In our C expression checker C32SAT [BB07] overflow detection is always automatically applied, which makes verification instances unnecessarily hard if we do not care about overflows. Thus, we separated overflow detection from arithmetic, and introduced an additional set of overflow detection operators in BTOR. We get only harder verification instances if we are interested in overflow detection.

Finally, it is of course possible to express some of our operators in terms of others. For instance our SMT solver Boolector [BB09a] supports the full set of operators, both in the application interface (API) and of course also in the parser for the BTOR format. Internally, however, we only use a subset of *base* operators, which are shown underlined and bold. Our selection can be considered to be arbitrary. It is motivated by the kind of word-level

class	operators	w_1	w_2	w_r
bitwise	**and**, or, xor, nand, nor, xnor	n	n	n
boolean	implies, iff	1	1	1
arithmetic	**add**, sub, **mul**, **urem**, srem **udiv**, sdiv, smod	n	n	n
relational	**eq**, ne, **ult**, slt, [us]lte, [us]gt, [us]gte	n	n	1
shift	**sll**, **srl**, sra, ror, rol	n	$log_2 n$	n
overflow	[us]addo, [us]subo, [us]mulo, sdivo	n	n	1
concatenation	**concat**	n_1	n_2	$n_1 + n_2$

Table A.2: Binary bit-vector operators. Some operators can be used in a signed or unsigned context, e.g. *sdiv* represents signed bit-vector division with two's complement semantics. For every arithmetic operator there is a corresponding overflow detection operator. The only exception is *udiv* as unsigned division can never overflow. Signed division overflows [BB07] if we divide the smallest negative integer by -1.

class	operators	w_1	w_2	w_3	w_r
conditional	**cond**	1	n	n	n

Table A.3: The only ternary bit-vector operator *cond* represents a functional if-then-else. If the condition is 1, it returns the second argument, and the third argument otherwise.

class	operators	w_1	upper	lower	w_r
extract	**slice**	n	u	l	u - l + 1

Table A.4: Miscellaneous bit-vector operators. The first operand identifies the variable to which *slice* is applied, *upper* and *lower* are immediates. The *slice* operator is the only operator that uses immediates.

simplifications we implemented in Boolector, which could be different for other solvers.

We can imagine various word-level techniques that can benefit from non-base operators. However, in contrast to the bit-level AIGER format it is much harder on the word-level to extract non-base operators after they have been represented with base operators. Therefore, we argue that in a general format for bit-precise word-level modelling it should be possible to express and retain non-base operators. On the other hand it is always possible to rewrite benchmarks that contain non-base operators into the base format.

A.3 Arrays

Arrays can be constructed with *array*. The first argument is the bit-width of the elements, and the second the size of the memory as power of two, i.e. the number of address bits. BTOR supports bit-vector arrays in combination with the array operations *read*, *write*, *acond* and *eq*, as the following example shows:

```
1 array 32 4          6 var 1                  11 eq 1 3 8
2 array 32 4          7 acond 32 4 6 1 2       12 and 1 10 11
3 array 32 4          8 write 32 4 7 4 5       13 root 1 12
4 var 4               9 read 32 8 4
5 var 32             10 eq 1 5 9
```

In the first three lines we declare three arrays with element bit-width 32, and size $2^4 = 16$. In line 7 we use an if-then-else on the arrays 1 and 2. If the condition 6 is true, we return 1, and 2 otherwise. The result is an array with bit-width 32 and size 16. Line 8 writes on 7 the element 5 at index 4. The result is an array, where the element at position 4 is overwritten with 5. The elements on all other positions remain the same. In line 9 we read 32 bits on 8 at index 4, i.e. we read the value that has been written at this index before. In line 10 we compare the read value with 5. In line 11 we compare the arrays 3 and 8. Finally, we assert 10 and 11, and declare the result as root.

A.4 Sequential Extension

BTOR supports modelling sequential and synchronous circuits with registers and memories. Registers are modelled with the help of *var* and *next*, and memories with *array* and *anext*. Variables without *next*, and arrays without *anext*, are treated as primary inputs, which are fresh for every clock cycle.

Registers are initialized to zero. Memories are uninitialized as this is nearly always the case for main memory in software and memory blocks in hardware. Consider the following example.

```
1 var 32          3 constd 32 5          5 next 32 1 4
2 var 32          4 xor 32 1 2
```

In line 5 we apply *next* to 1, which determines that 1 is a register. Variable 2 is an input as there is no *next* function for it. The *next* function for 1 is 4, i.e. in every cycle we apply *xor* to the current content of the register 1, and the primary input 2.

Additionally, BTOR supports modelling safety properties, which can be used for model checking. A safety property is represented by a boolean root. Multiple boolean roots are implicitly *disjuncted* as they typically represent different bad states.

A.5 Case Study

We present a case study where we show how BTOR, in particular its sequential extension, can be used to model hardware and systems. We use BTOR to model a FIFO with internal memory.

A.5.1 FIFOs

We took two different Verilog implementations of a typical hardware FIFO as shown in Fig. A.1, and modelled both with BTOR. In this case we manually translated the Verilog models to BTOR, but in principle this could be automated. The first implementation organizes its internal memory as stack, while the second implementation uses a queue. Both implementations use

Figure A.1: Hardware FIFO as black box. The clock signal is omitted.

head resp. tail registers that hold the address of the first and last element. The following Verilog code fragment shows how the queue-based implementation behaves if *dequeue* is active:

```
if (empty == 1'b0) begin
  data_out <= # 1 mem[head];
  head <= # 1 head + 3'b001;
end
if (head + 3'b001 == tail) begin empty <= # 1 1'b1; end
full <= # 1 1'b0;
```

Consider the following BTOR fragment for a queue-based FIFO implementation. It shows how the *next*-operator can be used to model internal memory.

```
1 var 1 reset          7 var 1 full          53 write 32 3 6 11 4
2 var 1 enqeue         8 var 1 empty         54 acond 32 3 7 6 53
3 var 1 deqeue         9 var 32 data_out     55 acond 32 3 2 54 6
4 var 32 data_in      10 var 3 head          56 acond 32 3 5 55 6
5 xor 1 2 3           11 var 3 tail          57 acond 32 3 1 56 6
6 array 32 3          ...                    58 anext 32 3 6 57
```

In line 58 the *next*-operator is applied to the array, which models internal FIFO memory. If *reset* is active (low), or *enqueue* and *dequeue* are equal, or *enqueue* is not active, or *enqueue* is active and the FIFO is full, memory remains in the same state. The if-then-else chain models the priority of the signals. However, if *enqueue* is active and the FIFO is not full, we write at the position the register *tail* points to the current input *data_in*. Analogously, we modelled the bit-vector registers *head*, *tail*, *empty*, *full* and *data_out*.

A.6 Experiments

We extended our SMT solver Boolector [BB09a] for checking sequential BTOR instances. In other words, we implemented a bounded model checker within Boolector. Boolector uses a functional AIG encoding including two level AIG rewriting [BB06]. Picosat [Bie08] is used as SAT solver. Boolector supports bounded model checking for witnesses [BCCZ99], and k-induction [SSS00] with and without All Different Constraints (ADCs). ADCs are used to represent simple path constraints. Note that model checking instances with memories generates ADCs with inequalities on arrays, which need extensionality [BB09a].

We compared Boolector's incremental model checking to non-incremental Boolector and Z3 1.2 [dMB08]. The benchmarks are parametrized instances for checking behavioral equivalence of two FIFO implementations as presented in A.5.1. Bit-width and size of memory is 32. The results are shown in table A.5. Column 1 shows the upper bound of the model checking instance. Column 2 shows the time of incremental model checking, which includes searching for witnesses and k-induction with ADCs. In column 3 ADCs are disabled.

The non-incremental results have been computed in the following way. First, we generated SMT instances that represent (*i*) search for witnesses, (*ii*) k-induction with ADCs, and (*iii*) k-induction without ADCs. In columns 4 and 6 we summed up the solving times for the results of (*i*) and (*ii*), from 0 to k. Analogously, we computed column 5 and 7 by summing up the results of (*i*) and (*iii*). Boolector clearly outperforms Z3. In the case where ADCs are disabled, the incremental model checking of Boolector is faster than the non-incremental.

	boolector				z3	
	inc		non-inc		non-inc	
k	adc	noadc	adc	noadc	adc	noadc
00	0.0	0.0	0.0	0.0	0.0	0.0
01	0.1	0.0	0.1	0.0	0.0	0.0
02	0.8	0.1	0.9	0.1	300.3	0.0
03	15.4	0.3	6.1	0.4	639.9	313.1
04	41.9	0.7	38.0	1.1	*to*	*to*
05	91.2	1.6	96.5	2.7	*to*	*to*
06	419.8	3.7	267.8	5.8	*to*	*to*
07	*to*	6.8	*to*	11.7	*to*	*to*
08	*to*	14.3	*to*	23.9	*to*	*to*
09	*to*	31.1	*to*	47.8	*to*	*to*
10	*to*	73.7	*to*	97.2	*to*	*to*

Table A.5: Experiments for equivalence checking of the FIFOs in A.5.1. We compare our SMT solver Boolector with Z3 1.2. We ran our benchmarks on our cluster of 3 GHz Pentium IV with 2 GB main memory, running Ubuntu Linux. Time limit is 900 seconds and memory limit is 1500 MB. Time out is indicated by *to*.

Appendix B

Detected Errors

In the following we show a selection of the most interesting errors that we encountered during our fuzzing experiments in section 4.4. The errors demonstrate that fuzzing was able to detect bugs deep in the core of SMT solvers. Error messages of the most common critical bugs that have been found, i.e. segmentation faults, aborts and floating point exceptions, are not shown.

Note that the random formulas that were used to test STP did not contain any division operators. Recall that all random formulas were syntactically valid and contained only supported operators and formula structures. However, some error messages misleadingly claimed that there were syntax errors. Finally, the error message shown for Z3 is from the emulator Wine[1]. Z3 caused a page fault.

Beaver 1.1 RC1

```
Fatal error:
exception Assert_failure("node_eval.ml", 876, 6)
```

```
Fatal error:
exception Assert_failure("div_simplification.ml", 345, 4)
```

```
Fatal error:
exception Assert_failure("circuit_generator.ml", 310, 6)
```

[1] www.winehp.org

```
Fatal error:
Aiger.create: cannot read file
"beaver-1.1-rc1-i686/share/beaver/strash_only/bvudiv_10.aig"

Fatal error:
exception Assert_failure("solver.ml", 84, 30)
```

CVC3 1.5

```
**** Fatal error in theory_core.cpp:325 (false)
Equivalence classes didn't get merged

**** Fatal error in search_sat.cpp:631 (false)
Should be unreachable

*** Unknown fatal exception caught

*** Fatal exception:
Soundness error: in bitvector_theorem_producer.cpp:2840
(e.getOpKind() == (isAnd ? BVAND : BVOR))
BitvectorTheoremProducer::andConst:
e = IF (0bin1 = (~(IF BVLT((IF (0bin1 =
IF BVLT(0bin1010011110010001,v1)
THEN 0bin1 ELSE 0bin0 ENDIF)
THEN 0bin1 ELSE 0bin0 ENDIF & (~(0bin0))),0bin0)
THEN 0bin1 ELSE 0bin0 ENDIF)))
THEN 0bin1 ELSE 0bin0 ENDIF

*** Fatal exception:
Illegal call to getExprValue()

*** Fatal exception:
Type Checking error:
Wrong bounds in bit-vector extract:
```

```
*** Fatal exception:
Type Checking error:
Mismatched bit-vector size in bit-wise AND (child #1).

*** Fatal exception:
Type Checking error:
Not a bit-vector expression (child #2) in bit-wise AND:
```

MathSAT 4.2.3

```
Floating point exception
[: 10: BV...: unexpected operator
```

OpenSMT

```
terminate called after throwing an instance
of 'std::invalid_argument'
  what():  mpq_set_str
```

Sateen SMT-COMP'08

```
UNSAT
unknown

UNSAT
sat
```

STP 0.1

```
Fatal Error:
TopLevelSAT: reached the end without proper conclusion:
either a divide by zero in the input or a bug in STP
```

Z3 1.2

```
wine: Unhandled page fault on read access to 0x00000004 at
address 0x4132c5 (thread 0009), starting debugger...
Unhandled exception: page fault on read access to 0x00000004
in 32-bit code (0x004132c5).
```

Appendix C

Curriculum Vitae

Short Biography

- 1980: born in Wels, Austria
- 1987 - 1991: Elementary School, Wels, Austria
- 1991 - 1996: High School, Wels, Austria
- 1996 - 2000: High School with emphasis on music, Grieskirchen, Austria
- 2000 - 2004: Bachelor student (Computer Science) at the Johannes Kepler University, Linz, Austria
- 2004 - 2006: Master student (Computer Science) at the Johannes Kepler University, Linz, Austria
- 2006 - now: Ph.D. student (Technical Sciences), and research and teaching assistant at the institute of Formal Models and Verification (FMV), Johannes Kepler University Linz, Austria

Awards

- Best Student Paper Award for "BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking" [BBL08], presented at BPR'08.
- Best SMT solver in `QF_BV` at SMT-COMP'08 [BDOS08].
- Best SMT solver in `QF_AUFBV` at SMT-COMP'08 [BDOS08].
- Second best SMT solver in `QF_BV` at SMT-COMP'09 [BDOS09].
- Best SMT solver in `QF_AUFBV` at SMT-COMP'09 [BDOS09].

Research Interests

- Satisfiability Modulo Theories (SMT)
- Formal Verification
- Fuzz Testing
- Delta-Debugging
- And-Inverter Graphs
- Propositional Satisfiability (SAT)

Teaching Activities

- Systems Programming
- Introduction into Programming
- Pthreads Programming
- Formal Modeling
- Model Checking

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, September, 2009

Dipl.-Ing. Robert Daniel Brummayer Bakk. techn.