A Constraint Solver and its Application to Machine Code Test Generation

Trevor Alexander Hansen

Submitted in total fulfilment of the requirements
of the degree of Doctor of Philosophy

October 2012

Department of Computing and Information Systems
The University of Melbourne
Australia

This thesis is printed on acid-free paper.

Abstract

Software defects are a curse, they are so difficult to find that most software is declared finished, only later to have defects discovered. Ideally, software tools would find most, or all of those defects for us. Bit-vector and array reasoning is important to the software testing and verification tools which aim to find those defects. The bulk of this dissertation investigates how to build a faster bit-vector and array solver.

The usefulness of a bit-vector and array solver depends chiefly on it being correct and efficient. Our work is practical, mostly we evaluate different simplifications that make problems easier to solve. In particular, we perform a bit-vector simplification phase that we call "theory-level bit-propagation" which propagates information throughout the problem. We describe how we tested parts of this simplification to show it is correct.

We compare three approaches to solving array problems. Surprisingly, on the problems we chose, we show that the simplest approach, a reduction from arrays and bit-vectors to arrays, has the best performance.

In the second part of this dissertation we study the symbolic execution of compiled software (binaries). We use the solver that we have built to perform symbolic execution of binaries. Symbolic execution is a program analysis technique that builds up expressions that describe all the possible states of a program in terms of its inputs. Symbolic execution suffers from the "path explosion", where the number of paths through a program grows tremendously large, making analysis impractical. We show an effective approach for addressing this problem.

Declaration

This is to certify that

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Trevor Alexander Hansen

Preface

The work discussing symbolic execution has been previously published as:

T. Hansen, P. Schachte, and H. Søndergaard[HSS09], "State Joining and Splitting for the Symbolic Execution of Binaries", Runtime Verification 2009.

Acknowledgements

Thanks first to my supervisors, Peter Schachte and Harald Søndergaard, for being patient—spending literally hundreds of hours working with me. And, for cheerfully improving my, initially appalling, writing and work. Thanks to Nicholas Nethercote, who supervised me during part of this work.

Thanks to my office mates: Ben Horsfall, for being good company and explaining logical formalisms; Thibaut Feydy, for knowing about constraint program; Marco S. Netto, for showing me how to be productive; Matt Davis, for being enthusiastic; and to the always friendly Khalid Al-Jasser, Kathryn Francis, and Jad Abi-Samra.

Thanks to Robert Brummayer and Armin Biere for making their SMT fuzzing and delta debugging tools available. We used them constantly during the development of our solver.

Thanks to Morgan Deters and to the other organisers of SMT-COMP 2010, 2011, and 2012. The competition has advanced the practice of building SMT solvers, and given us the chance to measure our progress against others.

Thanks to David Dill and Vijay Ganesh for making the STP solver, on which our bit-vector and array solver is based available.

Thanks to Zoe for allowing me to forgo large amounts of income, large amounts of leisure time, and large amounts of domestic duties, so that I could tap away at the computer.

I am grateful to Peter Stuckey who inspired our approach to solving array problems, and to Graeme Gange who suggested how to simply implement the approach in Minisat.

Thanks to the Australian Government for supporting me with a post-graduate scholarship.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

MILLIONS of computer programmers spend a substantial part of their days finding and fixing defects in their programs. Tools that make it easier to find defects in software are of enormous practical significance. This thesis contributes components that improve some of those defect-finding tools.

The bulk of the dissertation investigates approaches to efficiently solving bit-vector and array problems. The bit-vector theory introduces low-level operations such as multiplication and addition, which model the basic operations provided by a computer. These basic operations make it easy for software verification and testing tools to pose questions to such solvers about the effect of sequences of instructions.

The improvements to bit-vector and array solvers have made possible many tools. Amongst many uses, bit-vector and array solvers are used to automatically generate exploits for vulnerable software [ACHB11], to implement string solvers [GKA+11], to discharge theorems in theorem provers [BFSW11], and to check the equivalence of software [Smi11].

As an instance of a problem posed to a bit-vector solver, consider asking whether there exists a 64-bit value that when multiplied by 4 equals 12, but which is not 3. Because the arithmetic that computers perform can overflow, such a value exists.

The problem may be expressed in the SMT-LIB2 format as:

```
; This is a comment
(set-logic QF_BV); This says it's a bit-vector problem
(declare-fun x () (_ BitVec 64)); Makes a 64-bit variable

(assert (=
            (_ bv12 64)  ; The constant 12 (in 64 bits)
            (bvmul (_ bv4 64) x ) ; 4*x
        )
) ; 12 = 4*x (remember: not equivalent to 3 = x)

(check-sat)
; Prints x=3 (The first solution it happened to find)

(assert (not (= x (_ bv3 64) ))) ; x != 3
(check-sat)
; Prints x= 0x4000000000000003
```

Commands to the bit-vector solver are given between brackets. Anything to the right of a semi-colon is a comment. This creates a bit-vector x of 64 bits, then asserts to the bit-vector solver that $12 = (4 \times x)$. The `(check-sat)` command tells the bit-vector solver to look for a satisfying assignment to the problem. When we run this problem on our bit-vector solver, STP2, it reports that x=3 is such an assignment. However, it could have reported any of the possible assignments. Next, we assert to the bit-vector solver that $(x \neq 3)$, and ask for another satisfying assignment. This time it returns an assignment to x with the second-most significant bit set. This value when multiplied by 4 produced 12 as the result.

Bit-vector and array solvers take problems, usually from a software verification or testing tool, and decide whether satisfying assignments exist to those problems.

Our bit-vector solver STP2 is efficient; it won the `QF_BV` division at the annual SMT-COMP 2010. It placed second at the 2011 contest. STP2 and another of our bit-vector solvers placed second and third at the 2012 contest. Since 2007, bit-vector solvers have gone through a dramatic performance improvement. Comparing winners on the SMT-COMP 2007 benchmark set, the 2007 winner Spear v1.9 takes

3260s, the 2008 winner Boolector takes 1029s, and the 2009 winner MathSAT 4.3 takes 355 seconds. Our solver, STP2 r1659, solves the problems in 210 seconds.

STP2 is based on STP, an open-source solver that was equal winner at the 2006 contest. Modern bit-vector solvers contain hundreds of simplification and optimisation rules. We have made more than a thousand, sometimes small, changes to STP that cumulatively have the effect of making STP2 amongst the best available solvers.

Towards the end of this dissertation, we investigate an application of STP2 to the symbolic execution of machine code programs. *Symbolic execution* builds formulae that describe the value of program variables as functions of the program's inputs.

This thesis makes several significant contributions. In chapter 3, we identify simplifications that when combined give a bit-vector solver that is extremely efficient. In chapter 4, we describe a particular simplification which utilises bit-propagation to speed up bit-vector solving significantly. In chapter 5, we describe a novel decision procedure for solving problems in the combined theory of arrays and bit-vectors. Finally, in chapter 6 we describe an approach to improving test generation of binary programs.

2
Preliminaries

THIS chapter gives a quick review of the basic concepts that we rely on. The material we present here is not detailed enough to learn the concepts. Instead, this section is intended to refresh the reader's knowledge of the material and to fix the notation we use. In particular, we introduce the theories of bit-vectors and arrays. The solver that we spend the majority of this dissertation discussing solves satisfiability problems in these theories.

2.1 SAT Solving

A *propositional formula* is: true (1), false (0), a Boolean variable, the negation of a propositional formula, or the conjunction of two propositional formulae. For convenience, we add redundant propositional operations (Table 2.1). A classical *truth assignment* μ is a mapping from each of the Boolean variables in a propositional formula to either 1 or 0. Here 0 denotes falsehood and 1 denotes truth. So we write the fact that μ makes a Boolean variable b true by $\mu(b) = 1$. The *domain* $dom(\mu)$ of μ

Logical Operation	Description
$\neg p$	logical not (evaluates to 1 iff $(p \iff 0)$)
$p_0 \wedge p_1$	logical and (evaluates to 1 iff $(p_0 \iff 1)$ and $(p_1 \iff 1)$)
$p_0 \vee p_1$	logical or $\neg(\neg p_0 \wedge \neg p_1)$
$p_0 \oplus p_1$	logical exclusive-or $((p_0 \wedge \neg p_1) \vee (\neg p_0 \wedge p_1))$
$p_0 \iff p_1$	logical if-and-only-if $(p_0 \oplus \neg p_1)$
$p_0 \implies p_1$	logical implication $(\neg p_0 \vee p_1)$
$\mathrm{ITE}(p_0, p_1, p_2)$	logical if-then-else (ITE) $(p_0 \implies p_1) \wedge (\neg p_0 \implies p_2)$

Table 2.1: Logical operations of `QF_BV` & `QF_ABV`

is the set of variables that are mapped, by μ, to 0 or 1. We denote the set of variables in a propositional formula p by $\mathit{vars}(p)$.

When a propositional formula p is evaluated subject to an assignment μ, it evaluates to either 1 or 0, provided $\mathit{vars}(p) \subseteq \mathit{dom}(\mu)$. We sometimes apply a truth assignment μ to a whole formula p, writing $\mu(p)$, μ being defined by natural extension. A *satisfying* assignment to a propositional formula is one for which the formula evaluates to 1.

The propositional satisfiability problem *(SAT)* is to decide if there exists a satisfying assignment to a propositional formula. A *SAT solver* is a decision procedure for the SAT problem. Currently, most SAT solvers accept *conjunctive normal form (CNF)* as input. The basic building blocks of CNF are *literals*, a literal being either a Boolean variable, or the negation of a variable. (We sometimes talk of literals being assigned a value, meaning the underlying variable is assigned a value that causes that particular literal to be either 0 or 1.) A *clause* is a disjunction of literals. A formula is in CNF if it is a conjunction of clauses. The Cook-Levin theorem [Coo71] states that the SAT problem for CNF is $\mathcal{NP}$-complete, so finding an answer is not necessarily easy. Propositional formulae p and p' are *equisatisfiable* if, loosely, p is satisfiable iff p' is satisfiable. In most contexts where p and p' share variables, it is understood that, for p and p' to be considered equisatisfiable, satisfying truth assignments for the two must agree on the shared variables. More precisely:

$$\begin{aligned}&\forall\mu(\mu(p) = 0) \iff \forall\mu(\mu(p') = 0) \wedge \\ &\forall\mu\exists\mu'(\mu(p) = \mu'(p') \wedge (v \in \mathit{vars}(p) \cap \mathit{vars}(p') \implies \mu(v) = \mu'(v)))\end{aligned}$$

The widely-used Tseitin transformation [Tse83] converts an arbitrary propositional formula into an equisatisfiable CNF formula in linear time by adding a linear number of fresh variables. Note the encoding size is linear only as long as the logical operations have a linear size encoding, as in our case. The Plaisted and Greenbaum translation [PG86] often achieves a more compact CNF encoding.

Example 2.1

Consider the propositional formula: $(b_0 \wedge (b_1 \oplus b_2))$, where b_0, b_1 and b_2 are propositional variables. A satisfying assignment is $[b_0 \mapsto 1, b_1 \mapsto 0, b_2 \mapsto 1]$. The formula can be converted into CNF via the Tseitin transformation with the aid of a fresh variable b_3. As CNF it becomes: $b_0 \wedge b_3 \wedge (\neg b_1 \vee b_2 \vee b_3) \wedge (b_1 \vee \neg b_2 \vee b_3) \wedge (b_1 \vee b_2 \vee \neg b_3) \wedge (\neg b_1 \vee \neg b_2 \vee \neg b_3)$. Because the exclusive-or expression, in the context of the rest of the formula, must be 1, an approach like Plaisted and Greenbaum's omits some clauses and still produces an equisatisfiable result: $b_0 \wedge b_3 \wedge (b_1 \vee b_2 \vee \neg b_3) \wedge (\neg b_1 \vee \neg b_2 \vee \neg b_3)$. ■

We introduce some SAT solving concepts because we apply an approach to solving array problems (section 5.5) that is built into a SAT solver.

The SAT solvers we use are based on the DPLL algorithm [DLL62]. However, modern SAT solvers have tremendously improved upon the DPLL algorithm. The DPLL algorithm alternately propagates information, and performs *search* which selects variable assignments heuristically. The DPLL algorithm takes CNF format as input. It lifts from Boolean logic to ternary logic where along with 1 and 0 variables may be *unassigned*, meaning their value is unknown. *Unit propagation* assigns 1 to an unassigned literal in some clause if all the other literals in that clause have been assigned 0. If all the literals in a clause are assigned 0, or if a clause is empty, then a *conflict* has occurred because the assignment does not satisfy the formula. Modern SAT solvers use the partial assignment to perform "conflict driven clause learning". When a conflict occurs, they generate a *conflict clause* that summarises which assignments cannot occur together. The conflict clause is conjoined with the other clauses to prevent that combination of assignments from occurring again.

A simple SAT solving algorithm is given in Algorithm 2.1.

When unit propagation has stabilised, search is performed. The *decision level* is the number of variables assigned via search. A *trail* is a list of the variables that are assigned, both by search and unit propagation, together with the decision levels at which they were assigned. Using the trail, a *cancel* undoes the work performed beyond a given decision level.

The *two watched literals* technique [MMZ+01] speeds up unit propagation. Two literals in each clause are watched. When one of the two watched literals is assigned 0, a new unassigned literal in the clause is searched for. If one does not exist and the clause does not contain a 1, then the remaining literal must be 1. The technique is particularly useful because it is independent of backtracking: it does not break sophisticated backtracking techniques.

Algorithm 2.1 A simple SAT solving algorithm

```
Require: $p$, a propositional formula in CNF
 1: Create $dl \leftarrow 0$, the decision level
 2: Create $\mu$: assignments from variables to: $\{0, 1\}$
 3: while true do
 4:     Perform unit propagation
 5:     if a conflict occurred during unit propagation then
 6:         if $dl = 0$ then
 7:             return unsatisfiable
 8:         end if
 9:         Create a conflict clause $c$
10:         $p \leftarrow c \wedge p$
11:         Cancel assignments until $c$ is not in conflict
12:         Update $dl$
13:     else
14:         if some variable is not in $\mu$ then
15:             Set a variable not in $\mu$ to either 1 or 0
16:             Increment $dl$
17:         else
18:             return satisfiable
19:         end if
20:     end if
21: end while
22: return satisfiable
```

The SAT solvers we use are *incremental*. This means that after solving some CNF formula p_0, the work performed speeds up solving $p_0 \wedge p_1$, where p_1 is another CNF formula.

For a thorough treatment of SAT solving history, design, and practice, see Biere et al. [BHvMW09].

2.2 SMT

The satisfiability modulo theories (SMT) problem is to find a satisfying assignment to a first-order logic formula where some functions are interpreted in one or more theories. The bit-vector and array theories we consider are decidable. We do not allow quantifiers, so solve for a propositional combination of functions. SMT solvers combine the efficiency of propositional satisfiability solvers (SAT), with the ability to reason at a higher theory-level.

The bit-vector and array theories that we consider in this dissertation, are just two of the theories defined as part of the SMT-LIB [BST10] initiative.

$$
\begin{array}{rcl}
t^{[n]} & ::= & \mid\ t^{[n]} \circ_t t^{[n]} \\
 & & \mid\ -t^{[n]} \mid (\mathit{bvnot}\ t^{[n]}) \mid \mathit{ite}(p, t^{[n]}, t^{[n]}) \text{ (term if-then-else)} \\
 & & \mid\ (t^{[m]} :: t^{[n-m]}) \text{ (concatenation)} \\
 & & \mid\ t^{[m]}[i, j], \text{ where: } j \geq 0, i < m, i - j + 1 = n \text{ (extraction)} \\
 & & \mid\ ([0|1]^n)_2 \mid v \mid \mathit{sext}^{[n]}(t^{[m]}), m < n \text{ (sign extend)} \\
\circ_t & ::= & (\ \mathit{bvxor}\ ) \mid (\ \mathit{bvand}\ ) \mid (\ \mathit{bvor}\ ) \mid (+) \mid (\%_u) \text{ (unsigned remainder)} \\
 & & \mid\ (\%_s) \text{ (signed remainder)} \mid (\ \mathit{mod}_s\ ) \text{ (signed modulus)} \\
 & & \mid\ (\times) \text{ (multiplication)} \mid (\ll) \text{ (left shift)} \mid (\gg_l) \text{ (right shift)} \\
 & & \mid\ (\gg_a) \text{ (arithmetic shift)} \mid (\div_u) \text{ (unsigned division)} \mid (\div_s) \text{ (signed division)} \\
p & ::= & \mid\ t^{[n]} \circ_p t^{[n]} \\
 & & \mid\ p \oplus p \mid p \vee p \mid p \wedge p \mid p \iff p \mid \neg p \mid \mathit{ite}(p, p, p) \mid 0 \mid 1 \mid b \\
\circ_p & ::= & (<_u) \mid (<_s) \mid (\leq_s) \mid (\leq_u) \mid =
\end{array}
$$

Figure 2.1: A grammar for QF_BV. b ranges over a countably infinite set of propositional variables, and v ranges over a countably infinite set of (fixed bit-width) bit-vector variables. An 's' subscript means the operation interprets bit-vectors as signed integers, a 'u' subscript means bit-vectors are interpreted as unsigned integers.

2.3 Bit-Vectors

The QF_BV language is the first-order quantifier-free theory of fixed-width bit-vectors. A fixed-width bit-vector ($t^{[n]}$) is a vector of n bits. The bits of a bit-vector are indexed from 0 to $n-1$, and are written with the zeroeth bit on the right. We indicate extraction of a single bit from a bit-vector as $t^{[n]}[i]$, where i is a natural number between 0 and $n-1$ inclusive. Figure 2.1 gives a grammar for the QF_BV language; the names corresponding to the symbols are given in Table 2.2. Bit-vector terms are signedness agnostic, that is, neither signed nor unsigned. The semantics of some operations treats terms as being signed integers, others treat them as unsigned. Unsigned operations interpret the bit-vector $t^{[n]}$ as the natural number $\sum_{i=0}^{n-1}(2^i \times t^{[n]}[i])$, where $t^{[n]}[i]$ yields the integer value 0 if $t^{[n]}$'s i^{th} bit is zero, otherwise it yields 1. Signed operations use two's-complement to interpret bit-vectors as the integer $(-2^{n-1} \times t[n-1]) + \sum_{i=0}^{n-2}(2^i \times t[i])$.

We indicate binary literals in brackets with a subscript of 2. For example, $(10)_2$ corresponds to a 2-bit bit-vector that denotes the decimal constant 2.

Because the bit-vector operations can overflow, some bit-vector arithmetic operations return different results to their integer counterparts. For instance, both multiplication and addition are performed modulo 2^n, where n is the bit-width. Because the bit-width of the result is the same as the bit-width of the operands, bits in the result at position n or above are discarded. If the bit-width of the result of multiplication was twice the width of the operands, as in some formulations, then there would be no overflow and the result would be the same as for integer multiplication.

Example 2.2

Consider the multiplication $(3^{[2]} \times 3^{[2]}) = ((11)_2 \times (11)_2) = (01)_2 = 1^{[2]}$. There are two bits in $1^{[2]}$, the least significant is 1 (that is, $1^{[2]}[0] = 1$), and the most significant is 0 (that is, $1^{[2]}[1] = 0$). ■

The semantics of bit-vectors is similar to that of integers, but differs in some important cases.

Example 2.3

Some instances of the bit-vector arithmetic producing perhaps unexpected results are:

- $\frac{3}{3} = 1, \frac{4}{3} = 1, \frac{5}{3} = 1$ (truncating division).
- $86^{[8]} \times 3^{[8]} = 2^{[8]}$ (overflow).
- $((011)_2 >_s (111)_2) \not\equiv ((011)_2 + (001)_2 >_s (111)_2 + (001)_2)$ (overflow).
- $(2x = 2y) \not\equiv (x = y)$. The equivalence fails to hold because there is no unique multiplicative inverse for an even number modulo 2^n.
- $(3x = 3y) \equiv (x = y)$. There is a unique multiplicative inverse for odd numbers.

■

Comprehensive descriptions of the `QF_BV` and `QF_ABV` languages are downloadable from the SMT-LIB website [BRST08]. As of 2011, the SMT-LIB2 format has

Bit-vector Operation	Description
$(t^{[n]} \ll t^{[n]})$	Left shift
$(t^{[n]} \gg_a t^{[n]})$	Arithmetic right shift
$(t^{[n]} \gg_l t^{[n]})$	Logical right shift
$(t^{[n]} :: t^{[m]})$	Concatenation
$t^{[n]}[i,j]$	Extract
$(bvnot\ t^{[n]})$	Bitwise negation
$-t^{[n]}$	Unary minus
$(t^{[n]}\ bvand\ t^{[n]})$	Bitwise and
$(t^{[n]}\ bvxor\ t^{[n]})$	Bitwise exclusive-or
$(t^{[n]}\ bvor\ t^{[n]})$	Bitwise or
$(t^{[n]} + t^{[n]})$	Modulo addition
$(t^{[n]} \times t^{[n]})$	Modulo multiplication
$(t^{[n]} - t^{[n]})$	Modulo subtraction
$(t^{[n]} \div_u t^{[n]})$	Unsigned division
$(t^{[n]} \div_s t^{[n]})$	Signed division
$(t^{[n]}\ \%_u\ t^{[n]})$	Unsigned remainder
$(t^{[n]}\ \%_s\ t^{[n]})$	Signed remainder
$(t^{[n]}\ mod_s\ t^{[n]})$	Signed modulus
$(t^{[n]} <_s t^{[n]})$	Signed less than
$(t^{[n]} \leq_s t^{[n]})$	Signed less than equals
$(t^{[n]} >_s t^{[n]})$	Signed greater than
$(t^{[n]} \geq_s t^{[n]})$	Signed greater than equals
$(t^{[n]} <_u t^{[n]})$	Unsigned less than
$(t^{[n]} \leq_u t^{[n]})$	Unsigned less than equals
$(t^{[n]} >_u t^{[n]})$	Unsigned greater than
$(t^{[n]} \geq_u t^{[n]})$	Unsigned greater than equals
$ite(p, t^{[n]}, t^{[n]})$	Term if-then-else

Table 2.2: The bit-vector operations of `QF_BV`

replaced the older SMT-LIB format. In this thesis, we conform with SMT-LIB2 unless indicated.

Some notes about the symbols given in Table 2.2:

- The i and j used by the *extract* operation are natural numbers including zero. In the fixed-width formulation of bit-vectors, which we use, it is not possible to use arbitrary terms as i or j.

- The "arithmetic shift right" operation ($\gg_a$), copies the most significant bit of the first operand as the value is right shifted. The logical shift operation moves in zeroes to the left.

- There is a single multiplication operation ($\times$). When the bit-width of the operands and results is the same, as in the QF_BV formulation, signed and unsigned multiplication are equivalent.

- Unsigned division ($\div_u$) rounds towards zero. It never overflows, that is, it returns the same result as integer division with rounding towards zero. Signed division ($\div_s$) also rounds towards zero, but it overflows when the most negative value is divided by minus one.

- Unsigned remainder ($\%_u$) gives the remainder of division rounding towards zero. Signed remainder ($\%_s$) gives the remainder of division with rounding towards zero. The signed modulus ($\%_s$), which is rarely used, gives the remainder for division rounding towards negative infinity.

- All of the bit-vector operations are total. So, division by zero is acceptable. For convenience, we define: $(t \,\%_s\, 0) = t$, $(t \; mod_s \; 0) = t$, $(t \,\%_u\, 0) = t$, and $(t \div_u 0) = 1$. For $(t <_s 0)$ we define $(t \div_s 0) = -1$, and for $(t \geq_s 0)$, $(t \div_s 0) = 1$. This avoids the more complicated SMT-LIB2 semantics for division by zero. Although division by zero is defined, it is treated specially and is not introduced when solving problems that do not already contain it. So, contradictions will not be proved if division by zero is not initially present in the problem.

The complexity of the decision problem for QF_BV was recently shown by Kovásznai et al. [KFB12] to be non-deterministic exponential-time complete.

2.4 Arrays

An array is a map from bit-vectors of width n to bit-vectors of width n'. Alternatively we can think of an array as a list of 2^n values, indexed from 0 to $2^n - 1$. We sometimes annotate arrays with "type" information. For example, if a maps from bit-vectors of bit-width 2, to bit-vectors of bit-width 3, we write it as $a^{[2:3]}$. The *select* function is used to return the contents of a particular location. The *store* function creates a new array. The array operations are shown in Table 2.3.

The QF_ABV language extends QF_BV with single-dimensional arrays that are manipulated using the *select* and *store* functions. QF_ABV is the extensional theory

Array Function	Description
$ite(p, a_0^{[n:m]}, a_1^{[n:m]})$	array ITE
$select(a_0^{[n:m]}, t^{[n]})$	array read
$store(a_0^{[n:m]}, t_0^{[n]}, t_1^{[m]})$	array write

Table 2.3: Array operations of QF_ABV

of arrays, so allows equality between arrays. Our solver STP2 does not handle extensionality, so it only solves for a fragment of QF_ABV.

While arrays are single-dimensional, multi-dimensional arrays can be simulated by concatenating multiple indices together.

$store(a^{[n:m]}, t^{[n]}, t^{[m]})$ returns a new array the same as $a^{[n:m]}$, except that at index $t^{[n]}$, the value is $t^{[m]}$. Arrays only contain, and are indexed by, bit-vectors. So, the number of values that an array contains will always be a power of two. Arrays are total, so there are no out-of-bounds indices.

2.5 Term Rewriting

Term rewriting is widely used by bit-vector solvers to simplify expressions. Rewrite rules can apply theorems that the SAT solver might struggle to determine. Babić [Bab08] gives the example that the best SAT solvers cannot prove that $(a^{[12]} \times b^{[12]})$ = $(b^{[12]} \times a^{[12]})$ in reasonable time. For this reason, Babić ([Bab08] page 89) reports that the Spear solver has approximately 160 rewrite rules. Franzén ([Fra10] page 42) reports that MathSAT contains close to 300 rewrite rules.

A rewrite rule contains *term variables*, which match arbitrary expressions. A *subterm* of a term t is t itself or, if t is composite, a subterm of one of t's children. A rewrite rule transforms a term of some arbitrary type, to an equivalent (usually simpler) term of the same type. An equality can be transformed into a rewrite rule by treating its variables as term variables, and by orienting the equality somehow.

Given two terms t_0 and t_1, a rewrite rule $t_0 \triangleright t_1$ has the property that t_0 and t_1 are equal for any possible assignment, and $t_0 >_r t_1$, where $>_r$ is some pre-specified partial ordering on terms. *Matching* t_0 to t_1 is finding a substitution (σ) for the term variables in t_0 that make $t_0[\sigma]$ syntactically identical to t_1.

2.6 Structural Hashing

STP2, like most other solvers, avoids the creation of duplicate sub-expressions. A single sub-expression is created in the case that two or more expressions refer to the same sub-expression. Because an expression may have identical children, for instance $(x + x)$, structural hashing produces an acyclic directed multi-graph. However, in our work the distinction between multi-graphs and graphs is largely irrelevant. Instead, we consider an expression to be a directed acyclic graph (DAG) with labels on edges ordering the sub-expressions. Our expression DAGs have a single root expression, oriented so the leaves of the DAG are constants and variables. Structural hashing goes by various names and is used in many contexts, including as the hash-consing of Lisp.

Smith [Smi11] reports that a bit-vector and array theory representation of the Blowfish cryptographic algorithm has 6.9×10^{5186} nodes in the tree representation, and $220,639$ in the DAG representation. Applying structural hashing to some expressions is clearly essential.

2.7 Sharing-Aware Transformations

In a shared expression, that is, one that has been structurally hashed (section 2.6) and where expressions are shared, changes which when viewed locally decrease the global number of expressions, may, because of the sharing, actually increase the total number of expressions. Transformations that take such sharing into account are variously called DAG aware, graph aware, sharing aware, size preserving, or size reducing. We call such transformations *sharing aware*. We call a transformation, which may increase the total number of expressions, because it is ignorant of the effect of its transformations on shared expressions, *speculative*.

Example 2.4

Consider replacing the expression $-(v_0 + v_1)$ with $(-v_0 - v_1)$, where v_0 and v_1 are bit-vector variables. This pushes unary minus through bit-vector addition. Initially, there were four expressions: v_0, v_1, $(v_0 + v_1)$, and $-(v_0 + v_1)$. After the transformation, there are still four expressions: v_0, $-v_0$, v_1, and $(-v_0 - v_1)$. However, if the term $(v_0 + v_1)$ is shared, that is, it is the child of another term, and if $(-v_0 - v_1)$ is not shared, then the transformation removes a unary minus expression, but creates a unary minus and a binary minus term. In that context the transformation has caused a total increase of one binary minus. ■

2.8 Bit-Blasting

Bit-blasting reduces a problem, expressed in some theory, to propositional logic. For instance, in the QF_BV theory a multiplication between two 64-bit terms is expressed as one term. However, when it is bit-blasted to CNF, thousands of clauses are produced that contain many fresh propositional variables.

Example 2.5

An algorithm to bit-blast an n-bit addition is shown below. The algorithm assumes that the operands have already been converted to propositional formulae. It returns a formula which faithfully mimics bit-vector addition. This translates a single bit-vector term to propositional logic. The translation does not introduce any fresh variables; however, fresh variables will most likely be introduced during conversion to CNF.

```
Require: $p_0^{[n]}, p_1^{[n]}$: arrays of formulae to add
1: Create: $r$, an empty array of formulae
2: Create: $carry$, a variable of type formula
3: $carry \leftarrow 0$
4: for $i \in 0 \ldots (n-1)$ do
5:     $r[i] \leftarrow p_0[i] \oplus p_1[i] \oplus carry$
6:     $carry \leftarrow (carry \wedge p_0[i]) \vee (carry \wedge p_1[i]) \vee (p_0[i] \wedge p_1[i])$
7: end for
8: return $r$
```

■

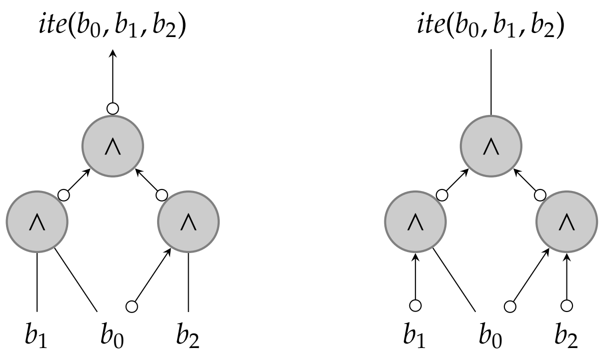


Figure 2.2: Two different AIGs corresponding to the propositional ITE. Hollow circles negate the value travelling along an edge.

2.9 And-Inverter Graphs

The bit-blasted propositional formulae that we create are stored as and-inverter graphs (AIGs). AIGs can store arbitrary propositional formulae in a non-canonical form, so there may be many possible distinct AIGs for a given propositional function. Figure 2.2 shows two AIGs for the propositional ITE. AIGs are useful to us because they give us structural hashing at the propositional level, and there are advanced approaches for manipulating and transforming them [BB04].

An AIG is a DAG where nodes correspond to logic gates, and the directed edges to wires that connect the logic gates. There are four types of nodes: the unique 1 node which has no incoming edges, *input* nodes which also have no incoming edges, *output* nodes which have at most one inward edge, and 2-input AND nodes. The edges may be inverted, complementing the result of the source node. As an AIG is built, structural hashing is performed so that there are no duplicates nodes. The 1 node may only be connected to *output* nodes.

Sharing aware simplifications [BB06] are applied at node creation time. The simple AIG structure makes it easy to apply local rewriting rules.

2.10 Propagators and Propagation Solvers

In chapter 4 we simplify bit-vector problems using an approach which is similar to a propagator based solver.

Finite-domain constraint problems have been studied in different fields of computer science, and different techniques have resulted. We shall make use of ideas developed in the field of artificial intelligence, namely Constraint Satisfaction Problems (CSPs) and propagation solvers.

Given a set of variables $x_1, \ldots x_k$, ranging over some set X (say, the set of integers), a CSP is a constraint C over the variables, together with a mapping D which associates, with each variable, a finite subset of X. In CSP terminology, the set $D(x)$ is the *domain* associated with x. A *constraint* is assumed to be in conjunctive form, that is, C is taken to be a conjunction of *primitive constraints*. The CSP (C, D) then represents the conjunctive constraint $C \wedge \bigwedge_{i=1}^{k} x_i \in D(x_i)$. In our use, domains will be integer intervals.

As an example, assuming that primitive constraints allow the use of linear arithmetic and inequality, we may have $C = (2 < x_1) \wedge (x_1 + x_1 < x_2)$, together with domains $D(x_1) = [1 \ldots 9], D(x_2) = [4 \ldots 8]$. The idea now is to use local reasoning rules to strengthen these constraints by narrowing the domains, without changing the set of possible solutions. For example, “node consistency” allows the use of the constraint $2 < x_1$ to narrow $D(x_1)$ to $[3 \ldots 9]$. “(Hyper-)arc consistency” can then use the constraint $x_1 + x_1 < x_2$ and simple interval arithmetic to narrow $D(x_2)$ to $[7 \ldots 8]$. Slightly more sophisticated reasoning can make use of parity information to determine x_2 completely.

We call hyper-arc consistent propagators *maximally precise propagators*.

More generally, each primitive constraint (schema) C has associated with it a set of *propagators*, *prop*(C). Each propagator may be able to narrow the domains of the constraint's variables. Formally, a propagator for C is a monotone function f operating on domains and satisfying $f(D) \sqsubseteq D$ (that is, f is decreasing). The propagator must preserve the set of solutions to $C \wedge D$. If we define $\mu \models D$ (“μ agrees with D”) to mean $\forall x \in dom(\mu)(\mu(x) \in D(x))$ we can state solution preservation more precisely: $\{\mu \mid \mu \models D \wedge \mu \models C\} = \{\mu \mid \mu \models f(D) \wedge \mu \models C\}$.

There is no requirement that a propagator is idempotent, that is, that $f(f(D)) = f(D)$. However, the idea behind propagation is that, given a CSP, we can apply a set of propagators, alternately and repeatedly, until no further domain improvement is possible. We follow Ohrimenko et al. [OSC09] and refer to the resulting idempotent function as a *propagation solver*.

These semi-formal definitions may leave the impression of a propagation solver as an unstructured “soup” of propagators. In practice we can make use of domain specific knowledge to impose restrictions on the order in which various propagators are employed, so as to use them most effectively. For example, we shall make use of nested solvers, that is, solvers that use propagators which are themselves fully-fledged propagation solvers.

Example 2.6

Consider $(v_0^{[5]} \times v_1^{[5]}) = v_2^{[5]}$, where each variable is known to be in an unsigned interval, initially: $D(v_0) = [1 \ldots 2]$, $D(v_1) = [1 \ldots 2]$, and $D(v_2) = [1 \ldots 6]$. There is no possible overflow, and a standard propagator for multiplication will shrink the domain of v_2 to $[1 \ldots 4]$. Note that, while there are assignments to v_0 and v_1 that allow v_2 to take values 1, 2, and 4, there is no assignment consistent with $v_2 = 3$. The strength of a propagator has to be measured with respect to properties of the domains used. For example, the interval domain cannot express membership of a set such as $\{1, 2, 4\}$. So, in spite of the information loss, the multiplication propagator is still considered *optimal*, as it produces the best possible interval. ■

3 Building a Better Bit-Vector Solver

SINCE the annual SMT solver competitions (SMT-COMP) ([BDdM+12]) began in 2005, there has been a dramatic improvement in the best bit-vector solvers' performance. In this chapter we describe the implementation of a high performance bit-vector solver called STP2. STP2 is open-source. The complete source code is available online from STP's source code repository.

Our solver is efficient; it won the `QF_BV` division at SMT-COMP 2010. However, few other solvers competed in 2010 because the input language's syntax had changed since the prior competition. STP2 placed second in the `QF_BV` division at SMT-COMP 2011. STP2 placed third in the `QF_BV` division at the SMT-COMP 2012. In section 3.22 we describe 4Simp which placed second in the `QF_BV` division at SMT-COMP 2012.

We show later (section 3.22) that (at least on the problems we have selected) the majority of STP2's success is due to just a few simplifications. We use the term *simplification* loosely to mean an equi-satisfiable transformation intended to make a problem faster to solve. In this chapter we describe about twenty different simplifications. However, we show that just a handful of those simplifications are really useful. This is an important result, highlighting where authors of bit-vector solvers should first focus their efforts. Of course, on different types of problems, the relative benefit of each simplification differs.

STP2 solves problems expressed in the `QF_BV` language described in section 2.3, a quantifier-free first order theory of fixed-width bit-vectors. This language is practically important because many software verification problems are expressible in it. We use the SMT-LIB2 bit-vector semantics, except that division by zero is defined differently.

In this chapter we focus on bit-vector problems. In chapter 4 we detail a particular bit-vector theory-level simplification. In chapter 5, we solve problems in a combined theory of arrays and bit-vectors.

3.1 STP 0.1 Overview

Vijay Ganesh and David Dill built the open-source STP 0.1 solver on which our STP2 solver is based. STP 0.1 is described in Vijay Ganesh's PhD thesis [Gan07], and a conference paper [GD07].

STP 0.1 converts bit-vector problems to CNF eagerly, and arrays lazily. If no array operations are used, then STP 0.1 acts as a compiler, converting bit-vector problems into CNF (section 2.1). The eager approach of bit-blasting problems is to create an equisatisfiable CNF encoding of the entire problem, which is then sent to a SAT solver. This reduces a bit-vector theory-level satisfiability problem to the propositional satisfiability problem (SAT). It contrasts with the lazy SMT approach [Seb07] which repeatedly switches between the SAT solver and a theory solver.

STP 0.1 has three main contributions. First, and most important, it showed that a well engineered bit-blasting bit-vector solver was competitive, and often superior, to other approaches. Second, it solved array problems via counter-example guided abstraction-refinement (which we discuss in chapter 5). Third, it used a partial solver in the simplification phase to determine some variables' bits' values (section 3.5).

3.2 STP2 Overview

STP2, like STP 0.1, is a bit-blasting bit-vector solver. Primarily STP2 differs from STP 0.1 in that extra simplification phases, or pre-solving, are applied; the simplifications are sharing-aware (section 2.7); and-inverter graphs (AIGs) (section 2.9) are used to hold the bit-blasted representation; and a more sophisticated CNF encoding approach is used.

STP2 can parse bit-vector and array constraints in the CVC3, SMT-LIB1, and SMT-LIB2 formats. It then simplifies them and encodes them via AIGs to CNF.

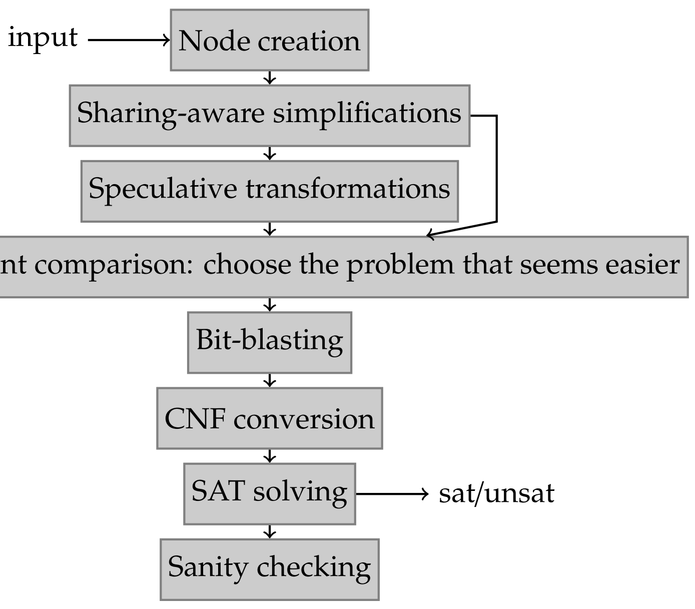

input
Node creation
Sharing-aware simplifications
Speculative transformations
Clause count comparison: choose the problem that seems easier
Bit-blasting
CNF conversion
SAT solving
sat/unsat
Sanity checking

Figure 3.1: Phases of STP2 when solving a `QF_BV` problem

STP2, like STP 0.1, encodes bit-vector constraints eagerly, and can encode array constraints lazily or eagerly.

STP2 preserves a copy of the input formula in memory, after structural hashing (section 2.6), constant folding and term normalisation. If the formula is satisfiable, as a check of its own correctness, STP2 substitutes the assignment found into the original formula. STP2 always maintains the book-keeping required to verify a satisfying assignment.

Broadly, STP2 simplifies constraints in two phases. In the first phase, simplifications that do not increase the number of expressions are applied. After they reach a fixed point, a copy of the formula and book-keeping is made. Next, speculative transformations are applied. These are transformations that could increase the total number of expressions, but which may simplify the problem drastically. Afterwards, if the resulting transformed problem seems more difficult than the result from the sharing-aware simplifications, then the expression is replaced with the previously saved expression (section 3.7).

In more detail, the phases of STP2, which are shown in Figure 3.1 are:

- *Parsing*: The input is parsed, and quick local transformations that simplify the problem are applied, as discussed in section 3.3.

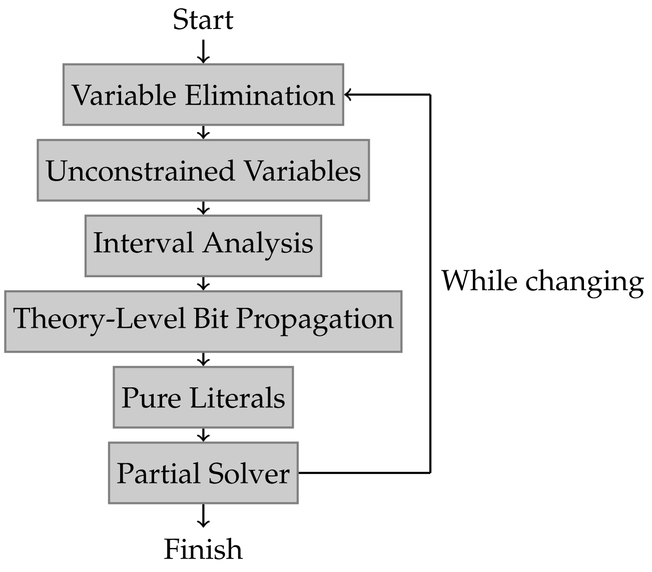


Figure 3.2: Sharing-aware simplifications performed by STP2. The sequence is repeated until they cause no change.

- *Sharing-aware simplifications*: Simplifications that do not increase the number of expressions are applied.
- *Speculative transformations*: Simplifications that may increase the number of expressions, such as distributing multiplication over plus are applied, as described in section 3.6.
- *Clause count comparison*: A quick estimate of the CNF size of the problem before and after speculative transformations is performed. The problem with the smaller estimated CNF size is chosen, as described in section 3.7.
- *Bit-blasting*: Convert to AIGs, as described in section 3.8.
- *CNF conversion*: Convert the AIGs into CNF.
- *SAT solving*: STP2 can use Cryptominisat [SNC09], Simplifying Minisat 2.2, or Minisat 2.2 [ES04] (the default) as a SAT solver. A DIMACS format CNF file can be output which most SAT solvers can parse.
- *Sanity checking*: If the SAT solver returns a model, as a check, evaluate the original formula with the model.

3.3 Simplifications when Creating Expressions

Before a new expression is created, *creation-time simplifications* transform the requested expression into an equivalent but potentially different expression. Creation-time simplifications make it impossible to create some expressions. For instance, it is impossible to create a term $t^{[n]} - t^{[n]}$. If such a term is requested, $0^{[n]}$ will be created instead. Many of the simplifications are highly specific, and will only apply occasionally. Their value is based on the fact that they are cheap to apply, and when applicable they may help tremendously. A principle of the creation-time simplifications is to produce few extra expressions. All creation-time simplifications create at worst a constant number of extra expressions, irrespective of the requested expression. STP2 applies more than 250 creation-time simplifications[1].

Example 3.1

The rule $((t_0 \,\%_u\, t_1) \gg_l t_0) \triangleright 0$ converts a term with an expensive remainder operation into a constant, resulting in a much smaller CNF encoding. If the arbitrarily complex terms t_1 or t_0 exist nowhere else, then they are eliminated from the problem. For the term $(t_2 \times ((t_0 \,\%_u\, t_1) \gg_l t_0))$, applying the rewrite rule will simplify it to $(t_2 \times 0)$, allowing the t_2 term to be potentially eliminated, too (if it is not referred to elsewhere). ■

The creation-time simplifications are idempotent; the simplifications are applied to the result before it is returned. That is, if any expression that is returned by the creation-time simplifications is requested, then the same expression will be returned. The following example will clarify this.

Example 3.2

If the term $((2 \times t^{[n]}) - (2 \times t^{[n]}))$ is requested, then the term $0^{[n]}$ is created. The $(t^{[n]} - t^{[n]})$ term, which is equivalent and simpler is not created. If $(t^{[n]} - t^{[n]})$ is requested, $0^{[n]}$ is created. The creation-time simplifications are idempotent, so requesting $((2 \times t^{[n]}) - (2 \times t^{[n]}))$ will not create $(t^{[n]} - t^{[n]})$. ■

[1]The *SimplifyingNodeFactory.cpp* file in STP's source-code repository contains the implementation.

Some rules have as their main purpose to normalise terms.

Example 3.3

The equivalent terms $(t^{[n]} + t^{[n]})$, $(2 \times t^{[n]})$ and $(t^{[n]} \ll 1)$ are all converted to $(2 \times t^{[n]})$. This kind of normalisation increases the chance that occurrences of equivalent expressions will be identified, so that rewrite rules like $(t^{[n]} - t^{[n]}) \triangleright 0$, will apply more frequently. ■

We do not aim to achieve a complete normalisation of equivalent terms, just commonly occurring ones. For example, one of the infinitely many equivalent but different terms, which is not converted to $(2 \times t^{[n]})$ is: $(t^{[n]}[n-2,0] :: 0^{[1]})$.

Some speculative transformations (section 2.7) are not applied at creation-time, owing to their potential to dramatically increase the number of terms.

Example 3.4

Consider $((x\ bvxor\ y)[u,l]) \triangleright (x[u,l]\ bvxor\ y[u,l])$, which can apply recursively. Applying this rule to the term $((t_0\ bvxor\ t_1)\ bvxor\ (t_2\ bvxor\ t_3))[4,2]$, gives:

$$((t_0[4,2]\ bvxor\ t_1[4,2])\ bvxor\ (t_2[4,2]\ bvxor\ t_3[4,2])).$$

In the worst case, the request for a single term has created seven terms (assuming the natural numbers 4, and 2 are free). Therefore, such rules are not applied at expression creation-time. ■

Example 3.5

When creating a bit-vector exclusive-or term, the following rules are applied, where $\prec$ is some fixed but arbitrary total order on terms:

$$(c_0\ bvxor\ c_1) \triangleright c_2, \text{ where constants } c_0 \text{ and } c_1 \text{ evaluate to } c_2 \quad (3.1)$$

$$(t_1\ bvxor\ t_0) \triangleright (t_0\ bvxor\ t_1), \text{ where } t_0 \prec t_1 \quad (3.2)$$

$$(t\ bvxor\ t) \triangleright 0 \quad (3.3)$$

$$(0\ bvxor\ t) \triangleright t \quad (3.4)$$

$$(-1\ \mathit{bvxor}\ t) \triangleright (\mathit{bvnot}\ t) \tag{3.5}$$

$$(t_0\ \mathit{bvxor}\ (\mathit{bvnot}\ t_1)) \triangleright (\mathit{bvnot}\ (t_0\ \mathit{bvxor}\ t_1)) \tag{3.6}$$

$$((\mathit{bvnot}\ t_0)\ \mathit{bvxor}\ t_1) \triangleright (\mathit{bvnot}\ (t_0\ \mathit{bvxor}\ t_1)) \tag{3.7}$$

■

We now describe different categories of simplification.

Constant Folding. Expressions that have only constant children are evaluated, and a constant is returned. For instance, instead of creating the $(6^{[5]} + 3^{[5]})$, or $(-3^{[5]} \times -3^{[5]})$ terms, the term $9^{[5]}$ is created. Equation 3.1 gives the rule for bit-vector exclusive-or constant folding.

Replacing Operations. The `QF_BV` language contains several similar operations. For instance, the language contains: unsigned less than, unsigned greater than, unsigned less than equals, and unsigned greater than equals. Using all of these would require duplicate code in the solver. So instead, we convert all the unsigned inequalities to unsigned greater than. Likewise the four signed inequalities are all replaced by signed greater-than. Some other operations removed are: not-and, not-or, Boolean-equals, Boolean implies, bit-vector rotate, and unsigned extension. These operations are compactly replaced by other operations.

Commutative sorting. The children of commutative operations are sorted. The children are put in three groups: constants, variables and other expressions. Each group is then ordered ($\prec$) based on a unique number that is allocated to an expression when it is created. Children are then ordered, starting with constants, then variables, then other expressions. Equation 3.2 is an instance of this normalisation.

Example 3.6

If the term $((\mathit{bvnot}\ t_1)\ \mathit{bvxor}\ (\mathit{bvnot}\ t_0))$ is requested where $t_0 \prec t_1 \prec (\mathit{bvnot}\ t_1) \prec (\mathit{bvnot}\ t_0)$, assuming the creation-time rules are checked top to bottom, then:

1. The children are already sorted, so no change is made.

2. Equation 3.6 is applied. The term $(\mathit{bvnot}\ ((\mathit{bvnot}\ t_1)\ \mathit{bvxor}\ t_0))$ is requested.

3. The bit-vector exclusive-or's children are sorted: $(\mathit{bvnot}\ (t_0\ \mathit{bvxor}\ (\mathit{bvnot}\ t_1)))$.

4. Equation 3.6 is applied. The term ($bvnot$ ($bvnot$ (t_0 $bvxor$ t_1))) is requested.

5. The simplified term (t_0 $bvxor$ t_1) is returned.

■

Rewrite rules. Rewrite rules are applied at creation-time. They have three different purposes. First we have rules that necessarily produce fewer terms. Second are rules that potentially increase the number of bit-vector terms by some fixed amount, but improve sharing. Third are rules that potentially increase the number of bit-vector terms, but produce an expression with a smaller CNF encoding.

The sharing aware rewrite rules return a sub-expression of the requested expression. Returning a reference to an existing expression requires no new expressions to be created. A similar idea is applied during creation of AIGs by Brummayer and Biere [BB06].

Example 3.7

Some instances of rewrite rules are:

- $((t^{[n]}\ bvxor\ t^{[n]}) \triangleright 0^{[n]})$. One term is requested, and at most one term is created (the $0^{[n]}$ term). This rule potentially increases the number of bit-vector terms, but improves sharing. Also, the resulting term might have a smaller CNF encoding. Whether the result of the rewrite rule actually has a smaller CNF encoding depends on whether later simplifications would have simplified it to zero anyway.

- $((bvnot\ t) \gg_l -t) \triangleright ite(t = 0, -1, 0)$. One term is requested, and at most four terms are created. However, the resulting term has a smaller CNF encoding.

- $((-1 \times -t) \triangleright t)$. One term is requested, but no extra terms are created because the rule returns a sub-expression of the requested expression.

■

Example 3.8

If the term (t_1 *bvxor* (*bvnot* t_0)) is requested, then both (*bvnot* t_0) and t_1 already exist. So returning the requested term will increase the total number of terms by at most one. If the term already exists, there will be no increase.

After applying the creation-time simplifications, in particular Equation 3.6, if $t_0 \prec t_1$ then (*bvnot* (t_0 *bvxor* t_1)) is returned. Both t_0 and t_1 already exist, but the *bvxor* term, and the *bvnot* term, might not, so the total number of terms is increased by at most two. ■

We found it advantageous to discover useful rewrite rules semi-automatically. In section 3.11 we discuss an approach to do that.

After applying the simplifications, structural hashing is performed which returns a reference to an existing expression if it already exists.

The creation-time simplifications achieve three goals. First, they eliminate equivalent but different expressions, making it easier to implement other simplifications. Second, eliminating equivalent but different expressions prevents those expressions from being encoded separately to CNF, which would necessitate extra SAT solver work. Third, simplifications can replace an expression with one that has a smaller CNF encoding.

These same simplifications could be applied in a separate simplification phase. However, applying them at creation-time means that the other simplifications reach a fixed point faster, and are easier to implement because they operate on fewer operations.

3.4 Variable Elimination

A variable can be eliminated from a problem if it is semantically equivalent to a term, and the term does not contain the variable.

Example 3.9

Consider the formula ($v_0^{[5]} = (v_1^{[5]} + 6^{[5]})$). Because v_0 is equal to a term which does not contain v_0, v_0 can be eliminated from the problem by replacing it with ($v_1^{[5]} + 6^{[5]}$) throughout. Alternatively, $v_1^{[5]}$ could be eliminated by replacing it with ($v_0^{[5]} - 6^{[5]}$).

After variable elimination, the formula becomes trivially satisfiable. ■

Some expressions that are not equalities have the same effect as an equality—of equating an expression and a variable. So, variable elimination is applied to more than just equalities.

Example 3.10

In the formula $((v_0^{[5]}\ \mathit{bvxor}\ 6^{[5]}) = 7^{[5]})$, v_0 can be eliminated from the problem by replacing it with $1^{[5]}$ throughout. After elimination the formula becomes trivially satisfiable. ■

Example 3.11

Consider the formula $((v_0 = 5) \vee ((v_0 + v_1) = (2 \times v_0)))$. Neither v_0 nor v_1 can be eliminated because they appear under a disjunction, so are not necessarily equivalent to a particular term. ■

Suppose we have that $v = t$, where t is a term. The variable v may be eliminated from the problem by replacing it throughout by the term it equals.

This occurs when an equation is conjoined at the top level, i.e. $((v = t) \wedge p)$, where the term t does not contain the variable v. Then, all occurrences of v in p are replaced with t. For correctness, the variable is replaced throughout the problem before other variables are eliminated.

Replacing the variables with expressions does not create a blow-up because a single shared expression replaces the variable in each sub-expression.

Variable elimination has four advantages. First, it reduces the number of variables in the problem that are functionally related to each other. Second, eliminating variables gives smaller CNF encodings because fewer equalities occur. Third, further theory-level simplification might become applicable. Fourth, if there are no other occurrences of the variable, then the expression it equals might disappear from the problem.

We store away the equivalences of eliminated variables so that later, if required, a model can be built.

Example 3.12

Consider the expression $((v_0 = -t) \wedge (v_1 = -v_0))$, where t contains neither v_0 nor v_1. Without variable elimination the CNF encoding contains clauses for two unary minuses, two equalities, and the t term. After variable elimination, the problem is simplified to 1, so the SAT solver is not called. If a model is needed, STP2 assigns zero to the variables in t, and evaluates t with this assignment to calculate the assignments to v_0, and v_1. ■

The variable elimination process is shown in Algorithm 3.1. It makes use of the procedures defined in Algorithm 3.2 and Algorithm 3.3. The algorithm attempts to isolate variables on the left-hand side of an equivalence, by iteratively moving expressions to the right-hand side. It generates some of the equivalences that are entailed by the original formula. Starting from the top-most expression, the algorithm finds candidates where a variable is equivalent to an expression. An elimination is allowed if the variable does not appear in the expression. Because of the normalisation that occurs when expressions are created, there is no need for the algorithm's pattern matching to be exhaustive. For instance, a bi-implication is not matched because it is converted to an exclusive-or at creation-time.

Replacing a variable by the term it equals will remove the equality expression. That is, given $(v = t)$, replacing v with t gives $(t = t)$ which is simplified to 1 by the creation-time simplifications. However, because other operations, such as bit-vector exclusive-or, do not reduce to 1 so readily, the variable elimination algorithm explicitly removes an expression which is used to eliminate a variable. This is safe because replacing a variable throughout by the expression it equals, makes the expression the equality was derived from redundant.

The variable elimination algorithm we give is not idempotent. For instance, given the expression $((v_1 \times v_0) = t) \wedge (v_1 = 1)$, no elimination is performed when traversing the left conjunct. On the right-hand side the elimination $v_1 = 1$ is generated. The result from variable elimination is: $v_0 = t$, from which v_0 might be eliminated if variable elimination is run again. We run the variable elimination algorithm as part of a series of simplifications (Figure 3.2) that are run until a fixed point.

Algorithm 3.1 The procedure for generating equivalences. This procedure is initially called with the top-most node of a problem.

```
Require: $e$                                                  // The current node
 1: procedure searchAnd($e$)
 2:     Create Boolean $changed \leftarrow 0$
 3:     if $e$ is a variable then
 4:         Eliminate with $e \iff 1$
 5:         $changed \leftarrow 1$
 6:     else if $e$ matches $(p_0 \wedge p_1)$ then
 7:         searchAnd($p_0$)
 8:         searchAnd($p_1$)
 9:     else if $e$ matches $(p_0 \oplus p_1)$ then
10:         $changed \leftarrow$ searchProp($e, 1$)
11:     else if $e$ matches $(\neg p_0)$ then
12:         $changed \leftarrow$ searchProp($p_0, 0$)
13:     else if $e$ matches $(t_0 = t_1)$ then
14:                         // Does not evaluate the second disjunct if the first is true
15:         $changed \leftarrow$ searchTerm($t_0, t_1$) $\vee$ searchTerm($t_1, t_0$)
16:     end if
17:     if $changed$ then
18:         Replace $e$ with 1
19:     end if
20: end procedure
```

Example 3.13

Consider applying the variable elimination algorithm (Algorithm 3.1) to $(v_0 = v_1) \wedge ((v_0 + 6) = (2 \times v_2))$. The algorithm calls itself recursively with each conjunct. Eventually *searchTerm*(v_0, v_1) is called. Because v_0 is a variable not contained in the right-hand side, v_0 is replaced throughout by v_1. Since the formula $(v_0 = v_1)$ was used to eliminate a variable, $(v_0 = v_1)$ is dropped from the problem. *searchTerm*$((v_1 + 6), (2 \times v_2))$ is called soon after, which in turn calls both *searchTerm*$(6, ((2 \times v_2) - v_1))$ and *searchTerm*$(v_1, ((2 \times v_2) - 6))$. In the second case, v_1 has been isolated and so will be eliminated. The original expressions have been replaced by 1, and the equations between variables stored so that models can be constructed if needed. ■

In the worst case, the algorithm has exponential running time in the number of expressions. However, on the problems used in the evaluation (section 3.19) the runtime is always reasonable. Because the encoding to CNF guarantees completeness and since this is an anytime algorithm, to limit the worst case cost, an upper

Algorithm 3.2 Sub-procedure for generating equivalences: dealing with terms. The *inverse* function gives the multiplicative inverse of an odd constant.

```
procedure searchTerm($lhs, rhs$)
    Create Boolean $changed \leftarrow 0$
    if $lhs$ is a variable and is not a sub-expression of $rhs$ then
        Eliminate with $lhs \leftarrow rhs$
        $changed \leftarrow 1$
    else if $lhs$ matches $-t$ then
        $changed \leftarrow$ searchTerm($t, -rhs$)
    else if $lhs$ matches ($bvnot\ t$) then
        $changed \leftarrow$ searchTerm($t, (bvnot\ rhs)$)
    else if $lhs$ matches ($t_0\ bvxor\ t_1$) then
        $changed \leftarrow$ searchTerm($t_0, (t_1\ bvxor\ rhs)$) $\vee$ searchTerm($t_1, (t_0\ bvxor\ rhs)$)
    else if $lhs$ matches ($t_0 + t_1$) then
        $changed \leftarrow$ searchTerm($t_0, (rhs - t_1)$) $\vee$ searchTerm($t_1, (rhs - t_0)$)
    else if $lhs$ matches ($t_0 \times t_1$) and $t_0$ is an odd constant then
                              // Creation-time simplifications ensure $t_1$ is not a constant
        $changed \leftarrow$ searchTerm($t_1, (inverse(t_0) \times rhs)$)
    end if
    return $changed$
end procedure
```

Algorithm 3.3 Sub-procedure for generating equivalences: dealing with propositions (no top-level).

```
procedure searchProp($lhs, rhs$)
    Create Boolean $changed \leftarrow 0$
    if $lhs$ is a variable and is not a sub-expression of the $rhs$ then
        Eliminate with $lhs \iff rhs$
        $changed \leftarrow 1$
    else if $lhs$ matches $p_0 \oplus p_1$ then
        $changed \leftarrow$ searchProp($p_0, (rhs \oplus p_1)$) $\vee$ searchProp($p_1, (rhs \oplus p_0)$)
    else if $lhs$ matches $\neg p_0$ then
        $changed \leftarrow$ searchProp($p_0, \neg rhs$)
    else if $lhs$ matches $(t_0 = t_1) \wedge bitwidth(t_0) = 1$ then
        $changed \leftarrow$ searchTerm($t_0, ite(rhs, t_1, (bvnot\ t_1))$)
        if $\neg changed$ then
            $changed \leftarrow$ searchTerm($t_1, ite(rhs, t_0, (bvnot\ t_0))$)
        end if
    end if
    return $changed$
end procedure
```

bound can simply be placed on the algorithm's number of iterations, or on running time.

Example 3.14

Consider the expression with four variables: $v_0 \ldots v_3$, where $S_{i,j}$ are syntactic/term variables, and S_r is the formula we create. For some k, construct:

$$\begin{aligned}
S_{0,0} &\leftarrow (v_0 + v_1)\\
S_{0,1} &\leftarrow (v_2 + v_3)\\
S_{0,2} &\leftarrow (S_{0,0} + S_{0,1})\\
S_{i,0} &\leftarrow (S_{i-1,0} + S_{i-1,2}), \text{ for } 0 < i \leq k\\
S_{i,1} &\leftarrow (S_{i-1,1} + S_{i-1,2}), \text{ for } 0 < i \leq k\\
S_{i,2} &\leftarrow (S_{i,1} + S_{i,0}), \text{ for } 0 < i \leq k\\
S_r &\leftarrow (S_{k,2} = 0)
\end{aligned}$$

For $k > 1$, variable elimination calls the *searchTerm* procedure (8×3^k) times. For $k = 11$, S_r has about 40 subterms and is equivalent to: $177147 \times v_0 + 177147 \times v_1 + 177147 \times v_2 + 177147 \times v_3 = 0$. The *searchTerm* procedure is called about 1.4 million times. ■

Checking that a variable is not contained on the right-hand side is a potentially expensive operation that variable elimination must perform. Some software verification benchmarks are initially thousands of levels deep. With each variable eliminated, the depth of the remaining terms might increase by as much as the depth of the term that replaces the eliminated variable. Our implementation performs caching to avoid repeated traversals of the same expression.

Example 3.15

We list some eliminations that are performed when the given expression is conjoined at the top level. The variable v is a bit-vector variable, and b is a propositional variable, v and p are not sub-expression of the other expressions.

- $(-v = t)$. Replace v by $-t$.
- $5 - (3 \times v) = t$. Replace v with $((1/3) \times -(t - 5))$.
- $\neg(b \oplus p)$. Replace b with the formula p.

- $(\neg b)$. Replace b with 0.
- $(p_0 \oplus (p_1 \oplus (b \oplus p_2)))$. Replace b with $\neg((p_0 \oplus p_1) \oplus p_2)$.

■

As an example of an equation that is left unprocessed, consider $(v = (6 \times (v + 1)))$. Nothing is done in this case, although it would be correct to generate the equation $v = (-1/5) \times 6$. This shows that more powerful variable elimination algorithms exist than the one that we have given.

Unlike MathSAT [Fra10], we have not experimented with replacing extracts from variables with the expressions they equal. For instance, given $(v[u, l] = t) \wedge p$, MathSAT will eliminate part of v. The partial solver that we describe in the next section will eliminate part of the variable in the special case when $l = 0$.

Variable elimination is widely applied by bit-vector solvers [Fra10, Gan07]. However, the algorithm that we have presented eliminates variables in situations that other algorithms do not. Our method is obvious enough that is has surely been applied in other contexts, although we believe its application to bit-vector solving is novel.

3.5 Partial Solver

Barrett et al. [BDL98] reduce the bit-width of variables and the number of variables in a problem with equalities of a certain form. In Section 3.2 of Barrett et al. [BDL98] they describe a linear solver for equations of the form:

$$c_0 \times v_0 + \ldots + c_k \times v_k + c = 0$$

Here the c's are constants, and the equation is at the top level. By using basic rules of algebra, expressions of the form $c_0 \times v_0$ are isolated on one side. If the constant c_0 is odd, both sides of the equality are multiplied by c_0's unique multiplicative inverse, isolating v_0. The variable v_0 is then eliminated from the problem.

If the constant c_0 is even, then it can be written as $c_0 = 2^l \times c'$, where c' is odd and $l > 0$. The equality is then replaced by two equalities:

$$0^{[l]} = (-c_1 \times v_1 - \ldots - c_k \times v_k)[l - 1, 0] \tag{3.8}$$

$$v_0[n-l-1,0] = (1/c') \times (-c_1 \times v_1 - \ldots - c_k \times v_k)[n-1,l] \tag{3.9}$$

Equations of the latter form are used in turn to eliminate part of v_0.

The "partial linear solver" [GD07] of STP 0.1 is more general; instead of restricting solving just to equations of variables, arbitrary terms can appear in equalities. For instance STP 0.1's partial solver can eliminate v, given $((7 \times v) + t) = 0$, where t is an arbitrary term not containing v. This necessitates a check that v is not contained in t. The partial solver of STP 0.1 is claimed to perform at most $O(m^2k)$ multiplications, where m is the number of equations, and k is the number of variables. However, the algorithm's total run time also includes time to check whether terms contain a given variable, and to re-normalise equations after variable elimination. These steps, which are necessary to run the algorithm, may consume significant time.

STP2 preserves the partial solver of STP 0.1. As above, variable elimination (section 3.4), eliminates a variable which has an odd coefficient. The variable elimination algorithm additionally isolates variables contained in expressions with the logical exclusive-or, bit-vector exclusive-or, single bit equality, and bit-vector not.

The partial solver, but not the variable elimination algorithm, solves for variables in equations of the form $c_0 \times v_0 \ldots + c_k \times v_k + c = 0$, where all the c's are even constants. Similarly to Barrett et al. [BDL98], the algorithm reduces this to two equisatisfiable equations, one of which has at least one odd coefficient. Also, the partial solver can eliminate parts of variables; for instance it uses $v[1,0] = t$ to remove the lowest two bits of v throughout the problem.

3.6 Speculative Transformations

STP2 applies speculative transformations until they reach a fixed point. We call them speculative (section 2.7) because they might increase the total number of expressions. They do not necessarily reduce the difficulty of the problem as measured by the size of the CNF encoding.

We have never found a problem for which the /spectrans do not reach a fixed-point. However, such problems may exist.

A common motivating example that speculative transformations simplify is:

$$(t_0 \times (t_1 + t_2)) \neq ((t_0 \times t_1) + (t_0 \times t_2))$$

which is false, but which in general is difficult for SAT solvers to establish.

STP2 uses approximately one hundred of STP 0.1's normalisation rules, which were substantially published when implemented in CVC Lite [GBD05]. The normalisations increase the number of expressions by at worst a polynomial amount. After applying the speculative transformations, to estimate whether the problem has become easier we apply clause count comparison (section 3.7), and then discard the changes if the problem seems harder.

Examples of the rules applied during the speculative transformations are:

- $(t_0 \times t_1)[u,l] \triangleright (t_0[u,0] \times t_1[u,0])[u,l]$
- $(t_0 \times (t_1 + t_2)) \triangleright ((t_0 \times t_1) + (t_0 \times t_2))$

STP2 has fewer speculative rules than STP 0.1. Some of the problems in the SMT-LIB benchmarks grow large when the rules of STP 0.1 are applied. In some cases, the growth exceeds the memory limit before the phase can be completed and the transformations undone. Only some of the simplifications that are applied during this phase give a worst case polynomial increase in node size. Others increase the number of expressions by a constant amount.

We describe only these simplifications as speculative. But there is no guarantee that the sharing-aware simplifications that we apply will make the problem easier to solve. In particular, it is possible for a simplification to produce a problem which is both smaller and harder for a SAT solver.

3.7 Clause Count Estimation and Comparison

The speculative transformations (section 3.6) sometimes make the problem larger. In the worst case, they can cause a polynomial size increase in the number of expressions. To estimate whether simplifications have made the problem easier, STP2 estimates the number of CNF clauses that bit-blasting a particular expression will produce. We call this *clause count estimation*. The number of clauses is estimated twice, once before and once after the speculative transformations. This allows STP2 to use whichever formula is estimated to produce the fewest CNF clauses. We use the estimated number of clauses as a proxy for the difficulty of solving a problem. So if, after speculative transformations, the estimated difficulty has increased, a copy of the problem saved prior to applying the speculative transformations is used instead. We call this *clause count comparison*. A similar, but more local, approach is taken by AIG Rewriting [BB04].

Some expressions are encoded into many more clauses than others. For instance, a 64-bit signed division operator introduces about 65,000 clauses (Table 3.8), but the bit-vector negation operation introduces none. STP2's clause estimator has a weighting for each operation that estimates the number of clauses generated for an operation of a particular bit-width.

The algorithm we employ is not sophisticated or particularly accurate. We have adjusted its parameters through trial and error. But as we show (section 3.20) it is successful.

This approach has the advantage that the problem can be transformed through a more difficult state, perhaps by distributing multiplication over addition, which later drastically simplifies. If only transitions which decreased the difficulty were allowed, then it would not be possible to transition through more difficult intermediate states.

A disadvantage of the approach is that if speculative transformations shrink one part of the problem, and complicate another part, then the change will be accepted or rejected in entirety, without considering the local effects.

A more precise means of measuring difficulty is to bit-blast the problem entirely before and after speculative transformations. We do this for small instances. However, some crafted problems, in particular, have CNF encodings that are expensive to generate. They require hundreds of millions of clauses before speculative transformations, and afterwards require far fewer. It is inefficient to bit-blast such large problems prematurely.

3.8 Bit-Blasting

Bit-blasting converts an expression into an equisatisfiable propositional formula. We use bit-blasting in three contexts.

- First, as a simplification step to identify theory-level expressions that are equivalent to each other or to constants. These equivalences are used to remove equivalent but different theory-level expressions.
- Second, as part of measuring whether other simplifications have made the problem easier or not (section 3.7).
- Third, and most importantly, as a precursor to CNF encoding.

We use the open-source ABC package [BM10] to build the AIGs that are created when bit-blasting. For each theory-level operation, we have built a procedure that requests AIGs that faithfully represent the semantics of each bit-vector and logical operation.

For bit-vector problems that are deemed easy by the clause estimator (section 3.7), we bit-blast the problem once during the simplification phase. Performing bit-blasting as part of the preprocessing has the advantage that it can be applied with other simplifications until a fixed point, rather than as a final stage.

Bit-blasting can discover that some expressions have a constant value. This allows the expression to be replaced with the respective constant. After requesting an AIG corresponding to an expression, we traverse the AIG vector checking whether each node is 1 or 0. If all the nodes are constants, then the theory-level expression can be replaced by the corresponding constant.

We also use the AIGs to find theory-level expressions that are different but equivalent. We call this *bit-blasting equivalence checking*. We do this by storing the relationship between vectors of AIG nodes and the theory-level expression that they represent. After bit-blasting, we iterate through the map looking for pairs of distinct expressions with the same AIG vector—these are equivalent. When equivalent expressions are discovered, one of the expressions is substituted throughout for the other expression. Using AIGs like this avoids the necessity of explicitly applying some word-level rewrite rules.

Example 3.16

Some bit-blasting simplifications identified by this stage are:

- $(t^{[5]} >_s (3^{[5]}\ bvor\ t^{[5]}))$ is replaced by 0.

- $(00001)_2 = (t^{[5]} \ll t^{[5]})$ is replaced by 0.
- $\neg(1^{[3]} >_u (0^{[2]} :: v^{[1]}))$ is replaced by $(1^{[1]} = v^{[1]})$ if that expression exists elsewhere.
- $((v^{[2]} \times v^{[2]}) >_s 3^{[2]})$ is replaced by 1.

■

However, the most important use of bit-blasting is to produce a propositional formula that, when converted to CNF, is fast to solve.

Using AIGs to store propositional formulae is good for two reasons. First, the AIGs simplify the formula as it is constructed, for instance, AIG creation time simplifications apply $(p \wedge p) \triangleright p$, where p is an arbitrary propositional formula. Second, because formulae are structurally hashed, sharing is performed between theory-level operations. For instance, the propositional formulae for the least significant bit of $(t_0\ \mathit{bvand}\ t_1)$, and $(t_0 \times t_1)$ are identical, so the AIG representing the bit is shared.

The particular encodings chosen for bit-blasting bit-vector operations considerably affect the solver's speed. However, fast encodings are often specific to the solver's implementation. Because of this, like other authors, we omit a detailed description of the particular encodings we chose[2]. An issue is that poor encodings can be "repaired" by later stages. Poor encodings make later stages, for instance, a CNF converter which can fix a poor encoding, look unreasonably good. As an indication of the care we took: for each bit-vector operation we implemented several encodings, for instance, 10 multiplication encodings and 8 division encodings. Then, we applied parameter optimisation (section 3.18) to select the best combination of those encodings.

3.9 Multiplication with Sorting Networks

In this section we describe an interesting approach to encoding the multiplication operation using sorting networks. Our results with the new approach are disappointing, as it is no faster than the standard approach of addition networks, but we present the idea in the hope that others might advance it.

[2]The source code file *Bitblaster.cpp* in STP2's source-code repository contains the encodings we chose.

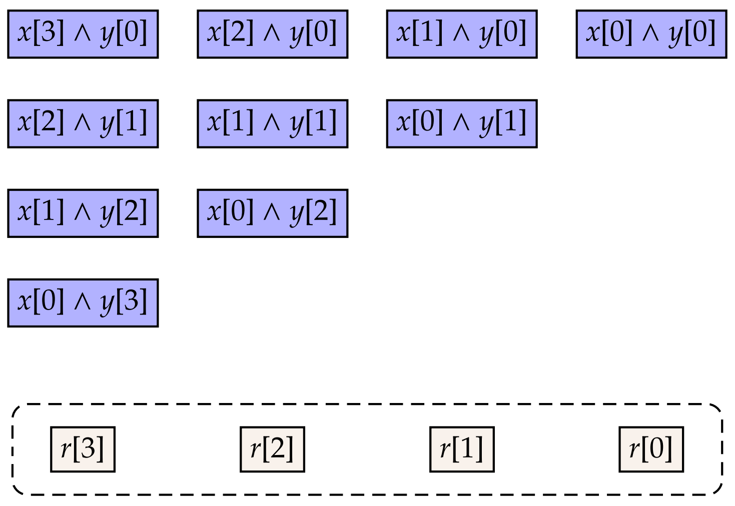


Figure 3.3: A 4-bit table of partial products. Column zero is on the right.

We encode multiplication using a table of partial products (Figure 3.3). This figure shows the table for a multiplication of $x^{[4]}$ and $y^{[4]}$, with the result $r^{[4]}$.

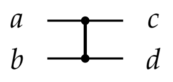
The exclusive-or of each column is taken to produce the result. For instance, the formula for the second least significant bit is: $r[1] = (x[0] \wedge y[1]) \oplus (x[1] \wedge y[0])$ Note, there is no $x[3] \wedge y[1]$ term used to calculate the value of r, which is ignored because it overflows.

A common approach to the summation of partial products is to capture the sum of a column by encoding it as some kind of addition network [ES06]. This corresponds to treating a column sum as a number in binary representation.

For 32 or 64-bit multiplication, there is a small upper bound on how large a column sum gets, even when carry-in is included. This makes it feasible to treat a column sum as a number in unary representation, which we had anticipated would lead to better constraint propagation. In this case summation becomes sorting and some kind of sorting network is called for in place of addition networks.

Given u inputs to a sorting network, a sorting network produces an output where, given ℓ true inputs, the outputs $s[0]$ to $s[\ell - 1]$ are 1, and the remainder $s[\ell]$ to $s[u - 1]$ are 0. Note that we want a sorting network to produce a bit sequence in non-increasing order, so the basic building block is the comparator shown in Figure 3.4.

To capture a sorting network we encode the method underlying Batcher's odd-even mergesort [Sed98]. Figure 3.5 shows the corresponding sorting network for



$c \leftrightarrow a \vee b$
$d \leftrightarrow a \wedge b$

Figure 3.4: A comparator and equations that describe it

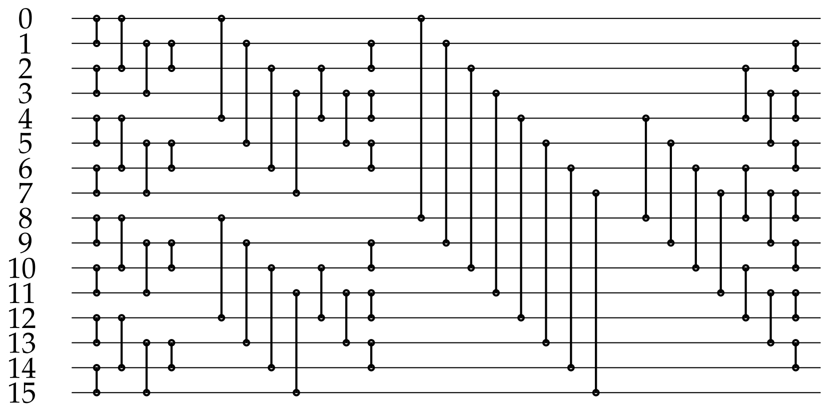


Figure 3.5: Batcher's odd-even sorting network (for 16 input bits)

16 input bits.

Batcher's algorithm is non-adaptive: merging is expressed in terms of compare-exchange operations only. As a consequence, the same sequence of operations happen irrespective of input. This makes the algorithm suitable for the purpose of CNF generation.

As carries into a column we use each second sorted value from the prior column. Batcher's sorting network is based on merging sorted values, so we are able to avoid re-sorting the carries which are already sorted. That is, we merge the sorted sequence $\{s[1], s[3], \cdots, s[u-1]\}$ with the sorted sequence of partial products for the next highest column.

The (binary representation) result r of the multiplication is now easily found: bit i should be the parity of the number of bits in column i. If there are an odd number of bits set, then the result is 1, if there are an even number the result is 0. So if u is even, the equation for the resulting bit is $(((s[0] \wedge \neg s[1]) \vee (s[2] \wedge \neg s[3]) \vee (s[4] \wedge \neg s[5]) \vee \ldots \vee (s[u-2] \wedge \neg s[u-1])) \Leftrightarrow r[i])$.

After encoding to CNF however, we did not find the sorting network encoding to be consistently faster than an addition network encoding. As we show later, in section 4.10, the unit propagation of the addition network encoding of multiplication is surprisingly strong.

3.10 CNF through Technology Mapping

In section 6 of their paper, Eén, Mishchenko, and Sörensson [EMS07] describe an encoding of AIGs to CNF, which they call *CNF through technology mapping* (TM). An implementation is available in the open-source ABC package [BM10].

STP2 uses ABC to manage AIGs and to convert those AIGs to CNF. ABC has two ways to convert AIGs to CNF. First, via a slightly improved [EMS07] Tseitin encoding, and second via the TM approach. STP 0.1 did not use AIGs, it used an implementation that we show (section 3.20) to be much less efficient.

The basic idea of the TM algorithm is to break the AIG into subgraphs of no more than *k* inputs, such that the sum of clauses needed to represent the subgraphs is minimised. Depending on their structure, subgraphs require differing numbers of clauses to represent them. For instance, an 8-input AIG denoting the "and" function can be represented with as few as 9 clauses, but many other functions require more. A partition of the AIG into subgraphs is chosen that heuristically minimises the number of clauses needed. The clauses that represent each possible subgraph are pre-generated via Minato-Morreales's algorithm [Min92].

Using ABC to manage AIGs, and to convert those AIGs to CNF, means that STP2 does not precisely control the clauses that are asserted. Some tools, for example Minisat+ [ES06], explicitly add extra clauses to the CNF to improve unit propagation. The extra clauses allows unit propagation to force entailments that are otherwise opaque and require *search* to discover. CNF representations which are maximally precise (section 2.10) under unit propagation have been created for many operations [Bac07]. We show later (Table 3.8) that ABC does a good job; it generates CNF that is maximally precise under unit propagation for some bitwise operations. However, the disadvantage is that we lose control over the CNF encoding, we cannot easily specify better encodings for particular operations.

3.11 Discovering Rewrite Rules

Initially, we discovered rewrite rules on an ad-hoc basis, implementing more than 100 rewrite rules. When STP2 solved a problem slowly, we looked through the problem for sub-expressions that could be simplified. However, to avoid missing important rules, we instead decided to automatically discover extra rewrite rules. In this section we refer to term rewriting concepts introduced in section 2.5.

We applied two approaches to generating rewrite rules. The first and simpler approach helped us to find useful rewrite rules. We generated all the equalities between a set of expressions and implemented as rewrite rules any that seemed reasonable. We automated just the process of discovering equivalences.

To discover equivalences, we automatically generate disequalities in a fragment of the `QF_BV` language. Then we send those disequalities to STP2, if STP2 reports that the disequality is unsatisfiable, then for that bit-width the equality is sound. If the equality holds, we then check whether the equality holds at some higher bit-widths. Of course, the equality might not hold at lower or higher bit-widths.

The second approach that we tried, but which unfortunately did not generate any useful rewrite rules, was more automated and generated rewrite rules rather than just equivalences. We give details of this in subsection 3.11.2. We were inspired to automatically generate rewrite rules by Bansal [Ban08], who generates rewrite rules to build a super-optimiser.

3.11.1 Finding Equivalences

In this section we give an approach to automatically discovering equivalent (but different) terms. We apply Algorithm 3.4 to discover equivalent expressions from a list of expressions. The algorithm is an optimised version of an all-pairs comparison. If the SAT solver discovers that two expressions' values differ for some assignment, then all the expressions in the list of expressions are evaluated with that assignment, splitting the list into at least two sub-lists.

Algorithm 3.4 has two procedures. The algorithm is started by calling the *discover* procedure. The *discover* procedure considers each pair of distinct expressions from the list. If the SAT solver discovers that there exists an assignment for which a pair of expressions evaluate to different values, then the algorithm calls the *split* procedure using that model. The algorithm widens the bit-width to check that the two expressions are equivalent on a range of bit-widths.

The *split* procedure splits the list into sub-lists where the terms in each list evaluate to the same value on an assignment. As the list is recursively split, the terms in each list have been shown to be equal at an increasing number of assignments.

Algorithm 3.4 Given a list of expressions, output expressions that are equivalent between the bit-widths of *low* and *minimum*. If time permits, check to a maximum bit-width of 1024. Calling *discover* checks all pairs of expressions contained in the list of expressions.

```
Require: list                        // a list of bit-vector terms of bit-width $n$
Require: assignment                  // a map from variables to integers
 1: procedure SPLIT(list, assignment)
 2:     Create newLists, a map from integers to lists of expressions
 3:     for $e \in list$ do
 4:         $newLists[eval(increase(e), assignment)].insert(e)$
 5:         // Evaluate the expression $e$ with the assignment. Place each expression
    that evaluates to the same integer into the same list. It might be necessary to
    increase the bit-width of $e$ to match the bit-width of the assignment.
 6:     end for
 7:     return newLists
 8: end procedure

Require: start                       // the bit-width to start testing expressions
Require: minimum                     // the least bit-width to test to
 9: procedure DISCOVER(list)
10:     for $i$ in $0 \ldots (size(list) - 1)$ do
11:         for $j$ in $0 \ldots (i - 1)$ do
12:             for $k$ in $start \ldots 1024$ do
13:                 if $(k > minimum) \land has\_timed\_out()$ then
14:                     break
15:                 end if
16:                 Changes the bit-width of $list[i]$ and $list[j]$ to $k$
17:                 if SAT_SOLVE($list[i] \neq list[j]$) is satisfiable then
18:                     Set assignment to be the model from the SAT solver
19:                     for all $l \in$ SPLIT($list, assignment$) do
20:                         DISCOVER($l$)
21:                     end for
22:                     return
23:                 end if
24:             end for
25:                 // The disequality is unsatisfiable at all the bit-widths tested
26:             Output $(list[i] = list[j])$          // Some duplicates output
27:         end for
28:     end for
29: end procedure
```

Note that Algorithm 3.4 splits using assignments at a range of bit-widths. If the expressions first differ at a bit-width of k, then the list will be split after temporarily increasing the bit-width of all the terms in the list to k bits.

We generated all subgraphs of the expression template given in Figure 3.6. There are three possible unary expressions: no-operation, unary minus, or bit-vector not.

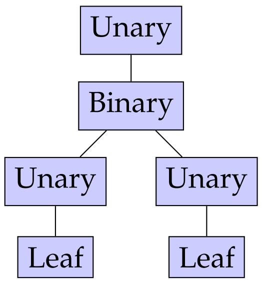


Figure 3.6: Expression instantiation template. We instantiate all possible subgraphs of this graph. There are 5 possible leaves, 3 possible unary operations, and 14 possible binary operations.

There are 14 different binary operations: addition, multiplication, bit-vector or, bit-vector and, bit-vector exclusive-or, leftshift, logical right shift, arithmetic right shift, subtraction, unsigned remainder, signed remainder, unsigned division, signed division, and signed modulus. We allow only five possible leaves: $v_0^{[n]}, v_1^{[n]}$, 0, 1, and -1. We generate expressions at a bit-width of 6, which allows a reasonable but not excessive number of constants. After applying the node creation-time simplifications (section 3.3) then removing duplicates, STP2 r1654 identifies 1570 unique expressions. Of these about 200 are equivalent at the bit-widths we tested. The rewrite rules that are not applied at creation time are mostly omitted because they might increase the total number of terms. Many of these are performed implicitly by other simplification phases.

Example 3.17

Two equivalences that are discovered, which are not amongst the creation-time simplifications are:

- $(1 \; bvxor \; t) \equiv (bvnot \; (-2 \; bvxor \; t))$
- $(t_0 + (bvnot \; t_1)) \equiv (bvnot \; (t_1 + -t_0))$

■

Some equivalences are expensive to verify. It takes about a minute to test at each bit-width between 6 bits and 19 bits for:

$$-(-1 \%_u v) = -((bvnot \; v) \%_u v)$$

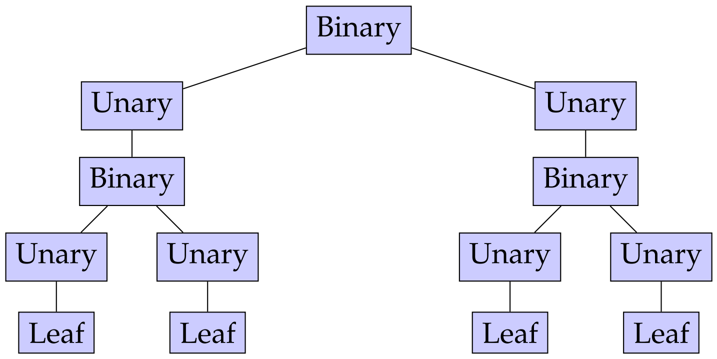


Figure 3.7: Expression instantiation template. We instantiate all possible subgraphs of this graph. Again, there are 5 possible leaves, 3 possible unary operations, and 14 possible binary operations.

3.11.2 Automatically Building a Rewrite System

Given the success in automatically generating equivalences, we decided to expand to automatically the work to convert equalities into rewrite rules. Informally, a rewrite rule is an equality that has been ordered so that it transforms a more complex expression into a simpler expression. We convert some equalities into rewrite rules.

To reduce the number of redundant rewrite rules, we rewrite all the subterms of the left and right-hand side of each rule. A term t is in normal form with respect to a set of rewrite rules, if no left-hand sides of any rewrite rule matches the subterms of t. A rule is *irreducible* if its left and right-hand sides are in normal form with respect to the set of rewrite rules (excluding itself). We produce only irreducible rules. For instance, given two rules $t_0 \triangleright t_1$ and $t_2 \triangleright t_3$, if $t_2[\sigma]$ equals t_0, then we replace t_1 with $t_3[\sigma]$.

The subgraphs of Figure 3.7 are input to the checking algorithm (Algorithm 3.4). We define $t_0 >_r t_1$ to hold in two cases:

- If t_0 is not a constant, but t_1 is a constant.
- If t_1 is a proper subterm of t_0.

If the ordering $>_r$ does not hold on the rewritten rule, the rule is removed. We use commutative matching to reduce the number of rewrite rules needed. With commutative matching every possible ordering of commutative operations' operands is considered when the matching is occurring. So for instance, the rule $((t + (\mathit{bvnot}\ t)) \triangleright -1)$ matches both $(v + (\mathit{bvnot}\ v))$ and $((\mathit{bvnot}\ v) + v)$.

We generate subgraphs of the template expression shown in Figure 3.7.

After applying the node creation-time simplifications (section 3.3) and the AIG equivalence simplification (section 3.8) then removing duplicates, there are about 30 million distinct expressions. We generate expressions at a fixed bit-width, currently 6 bits, and use sign-extension to increase the bit-width of constants.

It is not possible to apply an infinite sequence of rewrite rules to a term, because clearly, the number of sub-terms in the rewritten (the resulting) term strictly decreases.

After generating the rewrite rules, we test each equivalence at bit-widths from 6 bits to 1024 bits with a total timeout of 30 seconds. For instance:

$$((\mathit{bvnot}\ (-2 \div_s (\mathit{bvnot}\ v))) \gg_l (\mathit{bvnot}\ (v \ll -v))) = 0$$

holds for bit-widths from 6 to 63 bits, but it does not hold at a bit-width of 64. We do not test expressions for equivalence below a lower bound, currently 6 bits. We remove rules that do not hold at some bit-width.

We generate expressions and check if they are equivalent at some bit-widths using SAT via STP2. These problems are instances of the combinatorial equivalence checking problem, for which specialist solvers exist [KJJP09]. An alternative approach would be to combine some axioms of the `QF_BV` system to produce new theorems; this approach is not bit-width dependent.

Using Algorithm 3.4, we discovered about 120,000 rewrite rules in about 60 days of computer time. We stopped the algorithm before it finished. We show a selection of the rules that were discovered in Figure 3.8. Note that the number of rewrite rules needed is reduced significantly by the node creation-time simplifications. The rules that are generated are what we call sharing-aware (section 2.7). Expressions such as if-then-else expressions occur in the rewrite rules because some creation time transformations produce these expressions rather than the requested expression.

Running the rewrite rules on a random sample of 200 of the `QF_BV` benchmark set, no rules matched. So the rewrite rules we discovered are not useful for solving those problems.

In order to check that we did not miss important rewrite rules, we automatically generated a rewrite system. However, the rules we generated did not match any terms of our test problems. This shows that the common approach, of building rewrite systems by hand, produces good rewrite systems.

3.11.3 Summary

We found automatically generating equivalences useful to inspire new rewrite rules. For instance, initial versions of our creation-time simplifications omitted Equation 3.6. The equivalences we generated contained terms we realised would be simplified by the rule, so we implemented it.

In the first approach we manually inspected the proposed rules, and convinced ourselves that they are correct. In our second approach we recorded the bit-width intervals at which we tested each equivalence, and only applied the rewrite rule to expressions of a bit-width in that interval.

We had less success with automatically generating rewrite rules. The rules we generated applied occasionally to randomly generated problems but not to the evaluation problems. As shown in Figure 3.8, a large proportion of the rewrite rules contain constant values on the left hand side. As later work, omitting rules with constants on the left hand side might generate a more applicable rewrite system.

3.12 AIG Rewriting

When STP2 converts from a bit-vector theory formula to propositional logic, it stores the propositional formula as an AIG (section 2.9).

An approach to simplifying AIGs is to use *AIG rewriting* [BB04]. AIG rewriting performs local sharing-aware rewrites to the AIG, so that each rewrite does not increase the total number of AIG nodes. Representations with an equal number of nodes may contribute differently to the total number of nodes if one representation contains a subgraph that is used elsewhere in the AIG. Mishchenko et al. [MCB06], similarly to Bjesse et al. [BB04], measure the change in node count as functionally equivalent nodes replace each 4-input subgraph. Their rewriting is a greedy algorithm that reduces the AIG by iteratively replacing AIG subgraphs with equivalent but smaller pre-computed subgraphs, then selecting those replacements that reduce the total number of nodes. AIG rewriting is local, but its scope is enhanced by ap-

$$(-\mathit{ite}((111111)_2 = t_1^{[6]}, (000001)_2, (000000)_2) \gg_l (t_0^{[6]}\ \mathit{bvand}\ (\mathit{bvnot}\ t_1^{[6]}))) \quad \rhd \quad -\mathit{ite}((111111)_2 = t_1^{[6]}, (000001)_2, (000000)_2)$$

$$(-((111111)_2 \%_s\ t_1^{[6]}) \div_s \mathit{ite}((111110)_2 = t_1^{[6]}, (000000)_2, (000001)_2)) \quad \rhd \quad -((111111)_2 \%_s\ t_1^{[6]})$$

$$(-(t_0^{[6]} \ll (\mathit{bvnot}\ t_1^{[6]})) \%_s ((000001)_2 \div_u t_1^{[6]})) \quad \rhd \quad -(t_0^{[6]} \ll (\mathit{bvnot}\ t_1^{[6]}))$$

$$(-\mathit{ite}(((111110)_2 >_u t_1^{[6]}), (000000)_2, (000001)_2)\ \mathit{mod}_s\ -((111111)_2 \div_s (\mathit{bvnot}\ t_1^{[6]}))) \quad \rhd \quad (000000)_2$$

$$(-(t_1^{[6]} \ll t_0^{[6]})\ \mathit{bvand}\ (\mathit{bvnot}\ ((\mathit{bvnot}\ t_0^{[6]}) \div_s t_0^{[6]}))) \quad \rhd \quad (000000)_2$$

$$((\mathit{bvnot}\ (t_0^{[6]} \gg_l t_1^{[6]}))\ \mathit{bvand}\ -((\mathit{bvnot}\ t_1^{[6]}) \ll t_0^{[6]})) \quad \rhd \quad -((\mathit{bvnot}\ t_1^{[6]}) \ll t_0^{[6]})$$

$$((\mathit{bvnot}\ (t_0^{[6]}\ \mathit{bvor}\ (\mathit{bvnot}\ t_1^{[6]}))) \gg_l t_1^{[6]}) \quad \rhd \quad (000000)_2$$

$$(-\mathit{ite}((000001)_2 = t_1^{[6]}, (000000)_2, (000001)_2) \div_u -(-t_1^{[6]} \div_u t_1^{[6]})) \quad \rhd \quad \mathit{ite}((000001)_2 = t_1^{[6]}, (000000)_2, (000001)_2)$$

$$((\mathit{bvnot}\ ((111110)_2 \gg_l (\mathit{bvnot}\ t_1^{[6]})))\ \mathit{bvand}\ -(t_0^{[6]} \gg_l t_1^{[6]})) \quad \rhd \quad -(t_0^{[6]} \gg_l t_1^{[6]})$$

$$(\mathit{ite}((000000)_2 = t_1^{[6]}, (111111)_2, (000000)_2)\ \mathit{mod}_s\ (\mathit{bvnot}\ ((111110)_2 \gg_l t_1^{[6]}))) \quad \rhd \quad (000000)_2$$

$$(((000001)_2\ \mathit{bvxor}\ ((000001)_2\ \mathit{bvor}\ t_1^{[6]}))\ \mathit{mod}_s\ -((111110)_2 \gg_a t_0^{[6]})) \quad \rhd \quad (000000)_2$$

$$(-((000010)_2 \times t_1^{[6]})\ \mathit{mod}_s\ (\mathit{bvnot}\ ((00000)_2 :: t_0^{[6]}[0,0]))) \quad \rhd \quad (000000)_2$$

$$(\mathit{ite}((000000)_2 = t_1^{[6]}, (111111)_2, (000000)_2) \%_s ((\mathit{bvnot}\ t_1^{[6]}) \times (\mathit{bvnot}\ t_1^{[6]}))) \quad \rhd \quad (000000)_2$$

$$(-((\mathit{bvnot}\ t_0^{[6]}) \gg_a t_0^{[6]}) \gg_l -((111111)_2 \div_u t_1^{[6]})) \quad \rhd \quad (000000)_2$$

$$(-(t_1^{[6]} \gg_l t_0^{[6]}) \gg_l ((\mathit{bvnot}\ t_0^{[6]}) \gg_a (000001)_2)) \quad \rhd \quad (000000)_2$$

$$(((00000)_2 :: -t_1^{[6]}[5,5]) \gg_l \mathit{ite}((111110)_2 = t_1^{[6]}, (000000)_2, (111111)_2)) \quad \rhd \quad (000000)_2$$

$$((t_1^{[6]} \div_s (\mathit{bvnot}\ t_1^{[6]})) \ll -\mathit{ite}((000001)_2 = t_1^{[6]}, (000000)_2, (000001)_2)) \quad \rhd \quad (000000)_2$$

$$(-\mathit{ite}((111111)_2 = t_1^{[6]}, (000001)_2, (000000)_2) \%_u ((111110)_2 + -t_1^{[6]})) \quad \rhd \quad (000000)_2$$

$$(\mathit{ite}(((111110)_2 >_u t_1^{[6]}), (000000)_2, (000001)_2) \ll (\mathit{bvnot}\ ((\mathit{bvnot}\ t_1^{[6]}) \div_s (111110)_2))) \quad \rhd \quad (000000)_2$$

$$((\mathit{bvnot}\ (t_1^{[6]} \div_s (\mathit{bvnot}\ t_0^{[6]}))) \gg_l (\mathit{bvnot}\ (-t_1^{[6]} \gg_l -t_0^{[6]}))) \quad \rhd \quad (000000)_2$$

Figure 3.8: Twenty randomly selected rewrite rules that were automatically generated by Algorithm 3.4. Note that rules are given at a bit-width of 6, but can be safely widened to any larger bit-width for which they have been tested.

plying rewriting iteratively. We use the implementation from the open-source ABC tool [BM10].

Brummayer and Biere [BB06] give creation-time rules for AIGs that perhaps simplify, but never increase the total number of nodes. These rules are implemented in ABC, for instance, b is created instead of $(b \wedge b)$.

3.13 Boolean Abstraction AIG Rewrite

Observing that it is potentially expensive to apply AIG rewriting to the entire bit-blasted problem, we instead apply AIG rewriting just to the Boolean abstraction of a problem, sometimes called the propositional skeleton. We build an AIG that corresponds to the logical operations in the problem. Bit-vector theory predicates are replaced by fresh Boolean variables. Boolean variables map to themselves. This is the "Boolean abstraction of the input formula" used by DPLL(T) approaches ([Seb07] section 2.2).

Example 3.18

Given the expression $((v_0 = 0) \vee ((v_0 = 0) \wedge (v_1 = 1) \wedge (v_2 \leq_s 2))$, each of the predicates are mapped to fresh Boolean variables and substituted, giving $(b_0 \vee (b_0 \wedge b_1 \wedge b_2))$. AIG rewriting simplifies this to b_0, which after substitution back is $(v_0 = 0)$. ■

After abstraction, we apply AIG rewriting to the resulting AIG. Then, the AIG is converted back to a bit-vector theory expression, and the introduced Boolean variables are replaced with the predicates they substituted for. The intention is for the AIG rewrite to simplify the Boolean abstraction, perhaps removing theory-level expressions.

The system description for Boolector submitted to SMT-COMP 2012 mentions a top-level Boolean-skeleton simplifier, which we understand is an independent implementation of the idea in this section.

3.14 ITE Transformations

Kim et al. [KSJ09] simplify if-then-elses (ITEs) before applying a linear arithmetic solver. Their idea is that a term which is reachable via the "true" branch of an ITE can have any subterm that it shares with the ITE's conditional replaced by 1. Shared subterms that are reachable via the "false" branch can likewise be replaced by 0. We implement a variant of their approach, shown here as Algorithm 3.5.

The algorithm keeps a *context* of the conditions that must be 1 or 0 at a particular expression. Initially, at the root node, the context is empty. When we encounter an expression $ite(p, t_0, t_1)$, execution is forked, with p being added to the context

Algorithm 3.5 ITE simplification algorithm. Initially the procedure is called with the root node and an empty context.

```
procedure REPLACE_KNOWN($e, context$)
    if ($e \in context$) then
        return 1
    else if ($\neg e \in$ context) then
        return 0
    else if $size(context) > maximum\_size$ then
        return $e$                                    // Limit to prevent blowup
    else if $e$ matches $ite(p, t_0, t_1)$ then
        return $ite$(REPLACE_KNOWN($p, context$), REPLACE_KNOWN($t_0, context \cup p$), REPLACE_KNOWN($t_1, context \cup (\neg p)$))
    else
        Create $simplified$, a new expression
        Let the type of $simplified$ be the same as $e$
        for each child $c$ of $e$ do          // Add another child to the new expression
            $simplified.addChild$(REPLACE_KNOWN($c, context$))
        end for
        return $simplified$
    end if
end procedure
```

before t_0 is visited, and $\neg p$ before t_1 is visited. If a formula that is in the context is encountered, it is replaced by 1 or 0, as appropriate.

Example 3.19

Consider the expression $(ite(p_0 \wedge p_1, ite(p_0, v, 3), 5) = 5)$. $(p_0 \wedge p_1)$ is added to the context before the true branch of the outermost ITE is traversed, and $\neg(p_0 \wedge p_1)$ is added before the false branch is traversed. The condition of the innermost ITE, (p_0), is evaluated in the context $(p_0 \wedge p_1)$ and evaluates to 1, so it is replaced by 1. The simplified expression is: $(ite(p_0 \wedge p_1, v, 5) = 5)$ ■

This transformation just replaces formulae with 1 and 0. It does not do more elaborate transformations. For example, it leaves the expression $ite((t <_u 6), (t <_u 7), t = 6)$ unchanged, although it is equivalent to $(t \leq_u 6)$.

Example 3.20

As an example of the worst case behaviour, consider the following conjuncts, where S_0 is the root node, and S_1 to S_4 are syntactic variables:

$$\begin{aligned}
S_0 &= ite(p_0, (bvnot\ S_1), -S_1) \\
S_1 &= ite(p_1, (bvnot\ S_2), -S_2) \\
S_2 &= ite(p_2, (bvnot\ S_3), -S_3) \\
S_3 &= ite(p_3, (bvnot\ S_4), -S_4) \\
S_4 &= ite(p_4, v_0^{[n]}, v_1^{[n]})
\end{aligned}$$

Both v_0 and v_1 can be reached via the true and false branches of 4 ITEs. So there are 16 distinct contexts that can reach v_0, and 16 that can reach v_1. ■

The transformation is expensive because it considers the path through all the ITE nodes between an expression and the root node. That is, each extra ITE expression on the path from the root node doubles the number of node contexts. In the worst case, this creates 2^i contexts, where i is the number of ITE nodes. If a depth-first traversal is performed then space proportional to the depth of the ITE expressions is needed.

Other algorithms can find more substitutions than Algorithm 3.5 deduces. For instance, ROBDDs[Bry86] allow entailments that are missed by our algorithm to be deduced. However, such data structures are more expensive to maintain during traversal of the expression. Also, they are not perfect; like our algorithm they operate on the expression's Boolean abstraction.

3.15 Unconstrained-Variable Simplification

Bruttomesso [Bru08] and Brummayer [Bru09] both provide rules to simplify expressions that contain unconstrained variables. Some expressions containing unconstrained variables are eliminated by replacing them with fresh variables.

An *unconstrained variable* is one which has a single edge from a parent expression in the DAG representation of the problem. If the parent of an unconstrained variable can take any possible value, then the expression can be replaced by a fresh variable. Because the newly introduced fresh variable might also be unconstrained, the parent of the recently introduced fresh variable can sometimes be replaced by a fresh variable too.

Example 3.21

Consider $(v + 1)$, and assume this is the only use of v. Then, the occurrence of $(v + 1)$ can be replaced by a fresh variable. ■

The *unconstrained variable simplification* replaces an expression t with a fresh variable v. It can be applied if two conditions are satisfied: first, t can vary independently of the rest of the problem, and second, t denotes a surjective function, that is, t can yield any value in its range.

Example 3.22

Consider a sub-expression $v = t$, where v occurs nowhere else. This equation can be replaced by a fresh Boolean variable because the equality can evaluate to 1 or 0, that is, whatever is required to ensure the problem is satisfiable. If the equality must be 1, then v can be assigned t's value, otherwise it can be assigned something different from t, such as $(t + 1)$. ■

Example 3.23

Consider the expression $(v^{[1]} :: v^{[1]})$, where these are the only occurrences of the variable $v^{[1]}$. There are some values this expression cannot produce, for instance $(10)_2$. Even though v has only a single parent, it has two edges from that parent. So the expression cannot be replaced by a fresh variable. ■

Example 3.24

Brummayer [Bru09] considers the sub-expression $((v_0 + t) = (v_1 \textit{ bvand } v_2))$. Assume these occurrences of v_0, v_1, and v_2 are the only ones, while t is an otherwise arbitrary term. Because $(v_1 \textit{ bvand } v_2)$ can evaluate to any value, it can be replaced with a fresh variable v_3, giving $(v_0 + t) = v_3$. Because $(v_0 + t)$ can evaluate to any value, it too can be replaced by a fresh variable, giving $(v_4 = v_3)$. This can evaluate to either 1 or 0, so can be replaced by a fresh propositional variable b. The sub-expression can be 1 or 0, depending on what is required to make the problem satisfiable. ■

Bruttomesso [Bru08] describes the simplification applied to bit-vector problems. Brummayer independently developed the simplification, and gives the rules for the array variant, which he describes in section 3.4 of his thesis [Bru09]. Franzén [Fra10] gives the rules to build a model for the original problem from a model to the simplified problem. Following Franzén, STP2 keeps mappings between variables so that a model to the original problem can be calculated.

Most of the rules are straightforward; we give the rules in Table 3.1. Inequalities are complicated because of the possibility that they are necessarily 1, such as $v \geq 0$, or necessarily 0, such as $(v^{[3]} >_u 111)$, where v is unconstrained. Multiplication by a constant is complicated because only odd constants have a unique multiplicative inverse. Table 3.1 omits some of the operations in the language (section 2.3), because they are removed at creation-time.

In Table 3.1 the "Model" column gives the expressions that produce a model for the original problem given a model for the transformed problem. They produce *a* model, not *every* model.

STP2 also analyses the extracts from variables: if all of a variable's parents are extract expressions, and all of the extract expressions select different bits of the variable, then each extract expression is replaced by a fresh variable.

Example 3.25

Suppose a problem contains just two references to $v^{[20]}$, namely $v^{[20]}[15, 13]$ and $v^{[20]}[12, 2]$. Because the extracts do not overlap, and there are no other references to v, each extract could be replaced by a fresh variable. ■

Because applying the unconstrained simplification is based on the syntactic appearance of terms, only some equivalent terms will be simplified. The rules of Table 3.1 will for instance replace $ite(p, b_0, b_1)$, where p and b_1 are unconstrained, with a fresh variable. However, the different but equivalent expression $((p \implies b_0) \wedge ((\neg p) \implies b_1))$ will be left unchanged.

Expression	Condition	Replacement	Model
$(v_0^{[n]} :: v_1^{[m]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[m]})$	$v^{[n+m]}$	$v_0^{[n]} = v[m+n-1, m]$ $\wedge v_1^{[m]} = v[m-1, 0]$
$(bvnot\ v_0^{[n]})$	$unc(v_0^{[n]})$	$v^{[n]}$	$v_0^{[n]} = (bvnot\ v^{[n]})$
$-v_0^{[n]}$	$unc(v_0^{[n]})$	$v^{[n]}$	$v_0^{[n]} = -v^{[n]}$
$\neg b_0$	$unc(b_0)$	b	$b = \neg b_0$
$(v_0^{[n]} >_s v_1^{[n]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[n]})$ $\wedge n > 1$	b	$v_0^{[n]} = ite(b, 1, 0)$ $\wedge v_1^{[n]} = ite(b, 0, 1)$
$(v_0^{[n]} >_u v_1^{[n]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[n]})$ $\wedge n > 1$	b	$v_0^{[n]} = ite(b, 1, 0)$ $\wedge v_1^{[n]} = ite(b, 0, 1)$
$(v_0^{[n]} >_u v_1^{[n]})$	$unc(v_0^{[n]}) \wedge constant(v_1^{[n]})$ $\wedge v_1^{[n]} \neq max^{[n]}$	b	$v_0^{[n]} = ite(b, min^{[n]}, max^{[n]})$
$(v_0^{[n]} >_s v_1^{[n]})$	$unc(v_0^{[n]}) \wedge constant(v_1^{[n]})$ $\wedge v_1^{[n]} \neq max^{[n]}$	b	$v_0^{[n]} = ite(b, min^{[n]}, max^{[n]})$
$(v_0^{[n]} >_u v_1^{[n]})$	$unc(v_0^{[n]})$	$b \wedge (v_1^{[n]} \neq max^{[n]})$	$v_0^{[n]} = ite(b, max^{[n]}, min^{[n]})$
$(v_0^{[n]} >_s v_1^{[n]})$	$unc(v_0^{[n]})$	$b \wedge (v_1^{[n]} \neq max^{[n]})$	$v_0^{[n]} = ite(b, max^{[n]}, min^{[n]})$
$(v_0^{[n]} >_u v_1^{[n]})$	$unc(v_1^{[n]})$	$b \wedge (v_0^{[n]} \neq min^{[n]})$	$v_1^{[n]} = ite(b, min^{[n]}, max^{[n]})$
$(v_0^{[n]} >_s v_1^{[n]})$	$unc(v_1^{[n]})$	$b \wedge (v_0^{[n]} \neq min^{[n]})$	$v_1^{[n]} = ite(b, min^{[n]}, max^{[n]})$
$b_0 \wedge b_1$	$unc(b_0) \wedge unc(b_1)$	b	$b_0 = b \wedge b_1 = b$
$b_0 \vee b_1$	$unc(b_0) \wedge unc(b_1)$	b	$b_0 = b \wedge b_1 = b$
$(v_0^{[n]}\ bvand\ v_1^{[n]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[n]})$	$v^{[n]}$	$v_0^{[n]} = v^{[n]} \wedge v_1^{[n]} = v^{[n]}$
$(v_0^{[n]}\ bvor\ v_1^{[n]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[n]})$	$v^{[n]}$	$v_0^{[n]} = v^{[n]} \wedge v_1^{[n]} = v^{[n]}$
$b_0 \oplus b_1$	$unc(b_0)$	b	$b_0 = b_1 \oplus b$
$(v_0^{[n]}\ bvxor\ v_1^{[n]})$	$unc(v_0^{[n]})$	$v^{[n]}$	$v_0^{[n]} = (v_1^{[n]}\ bvxor\ v^{[n]})$
$ite(b_0, v_0^{[n]}, v_1^{[n]})$	$unc(b_0) \wedge unc(v_0^{[n]})$	$v^{[n]}$	$b_0 \wedge (v_0^{[n]} = v^{[n]})$
$ite(b_0, v_0^{[n]}, v_1^{[n]})$	$unc(b_0) \wedge unc(v_1^{[n]})$	$v^{[n]}$	$\neg b_0 \wedge (v_1^{[n]} = v^{[n]})$
$ite(b_0, v_0^{[n]}, v_1^{[n]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[n]})$	$v^{[n]}$	$(v_0^{[n]} = v^{[n]}) \wedge (v_1^{[n]} = v^{[n]})$
$(v_0^{[n]} \div_u v_1^{[n]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[n]})$	$v^{[n]}$	$v_0^{[n]} = v^{[n]} \wedge v_1^{[n]} = 1$
$(v_0^{[n]} \times v_1^{[n]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[n]})$	$v^{[n]}$	$v_0^{[n]} = 1 \wedge v_1^{[n]} = v^{[n]}$
$v_0^{[n]} = v_1^{[n]}$	$unc(v_0^{[n]})$	b	$v_0^{[n]} = ite(b, v_1^{[n]}, v_1^{[n]} + 1)$
$(v_0^{[n]} + v_1^{[n]})$	$unc(v_0^{[n]})$	$v^{[n]}$	$v_0^{[n]} = v^{[n]} - v_1^{[n]}$
$(v_0^{[n]} \gg_l v_1^{[n]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[n]})$	$v^{[n]}$	$v_0^{[n]} = v^{[n]} \wedge v_1^{[n]} = 0$
$(v_0^{[n]} \gg_a v_1^{[n]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[n]})$	$v^{[n]}$	$v_0^{[n]} = v^{[n]} \wedge v_1^{[n]} = 0$
$(v_0^{[n]} \ll v_1^{[n]})$	$unc(v_0^{[n]}) \wedge unc(v_1^{[n]})$	$v^{[n]}$	$v_0^{[n]} = v^{[n]} \wedge v_1^{[n]} = 0$

Table 3.1: Rules for the unconstrained variable simplification. *unc* is a predicate returning 1 iff its operand is a variable and is unconstrained. *v* is a fresh variable. The table lists one rule for some commutative operations (e.g. xor); for these operations the rules are applied with the operands reversed, too. Starting from the top, use the first rule that matches the expression. "*min*" and "*max*" are the signed or unsigned minimum and maximum values for the appropriate bit width, respectively. We give the rules only for term ITEs, they are similar for propositional ITEs.

There are cases which a more sophisticated algorithm could simplify. For instance, assume v does not occur elsewhere and consider: $ite(p, (v + t) = 6, v = 4)$,

$((v \div_u 10) <_u 2)$, or $((v < 10) \wedge (v > 2))$. All of these expressions denote surjective functions, so can be replaced by a fresh variable.

3.16 Pure Literal Elimination

We perform pure literal elimination to identify Boolean variables that can be set to a constant. Initially, this approach was called the affirmative-negative rule [DP60]. Pure literal elimination over graphs has also been called monotone input reduction [JBH10]. Algorithm 3.6 calculates the polarities for each expression in a problem. This algorithm replaces any Boolean variable that has a polarity of TRUE with the 1 expression, and any Boolean variable with a FALSE polarity by the 0 expression.

3.17 Interval Analysis

STP2 performs a bottom up unsigned interval analysis. We use standard rules to derive the bounds of operations. The bounds for logical operations are given by Warren [War02], the bounds for integer arithmetic operations are standard and can be found in constraint programming textbooks, e.g., Marriott and Stuckey [MS98]. During the analysis, any expression that has the same lower and upper bound is replaced by the corresponding constant expression.

In chapter 4 we investigate a more sophisticated variant of this.

The interval analysis is fast and imprecise. We do not take care to ensure that the bounds produced are as tight as possible. The outline of the algorithm that we use, and the implementation for some of the operations is given as Algorithm 3.7.

3.18 Parameter Optimisation

SMT solvers, like other decision procedures, often have many configuration options. For instance Z3 version 3.0 has 284 configuration options [dMP12]. STP2 has fewer, perhaps 30 that can be changed via the command line, but there are dozens more in the source code.

Parameter optimisation aims to find a good assignment to the configuration options of a decision procedure on some set of problems. To decide which simplifications to enable, we applied the parameter optimisation tool ParamILS [HBHH07,

Algorithm 3.6 Pure literal elimination. REPLACE is called initially with the root expression. It replaces any Boolean variable with a *TRUE* polarity by 1, and *FALSE* by 0. The possible polarities are *TRUE*, *FALSE*, and *BOTH*.

```
Require: $e$                                                // The current expression
Require: $current$                       // The polarity of the current expression
Require: $pol \leftarrow \{\}$                  // Maps from an expression to its polarity
 1: procedure CALCULATE_POLARITY($e, current, pol$)
 2:     if $pol[e] = TRUE \wedge current \neq TRUE$ then
 3:         $pol[e] \leftarrow BOTH$
 4:     else if $pol[e] = FALSE \wedge current \neq FALSE$ then
 5:         $pol[e] \leftarrow BOTH$
 6:     else if $pol[e] \neq BOTH$ then
 7:         $pol[e] \leftarrow current$
 8:     end if
 9:     if $e$ matches $(p_0 \wedge p_1)$ then
10:         CALCULATE_POLARITY($p_0, current, pol$)
11:         CALCULATE_POLARITY($p_1, current, pol$)
12:     else if $e$ matches $(p_0 \vee p_1)$ then
13:         CALCULATE_POLARITY($p_0, current, pol$)
14:         CALCULATE_POLARITY($p_1, current, pol$)
15:     else if $e$ matches $(\neg p)$ then
16:         if $current = BOTH$ then
17:             CALCULATE_POLARITY($p, BOTH, pol$)
18:         else if $current = TRUE$ then
19:             CALCULATE_POLARITY($p, FALSE, pol$)
20:         else
21:             CALCULATE_POLARITY($p, TRUE, pol$)
22:         end if
23:     else
24:         for each child $c$ of $e$ do
25:             CALCULATE_POLARITY($c, BOTH, pol$)
26:         end for
27:     end if
28: end procedure

29: procedure REPLACE($e$)
30:     Create $pol$, a map from expressions to their polarity
31:     CALCULATE_POLARITY($e, TRUE, pol$)
32:     for all $(b, polarity) \in pol$ do            // Iterate over all Boolean variables
33:         if $polarity = TRUE$ then
34:             Eliminate from $e$, with $b \iff 1$
35:         else if $polarity = FALSE$ then
36:             Eliminate from $e$, with $b \iff 0$
37:         end if
38:     end for
39: end procedure
```

HHLBS09]. ParamILS performs a hill-climbing search with random restarts to select a good, but probably not optimal, combination of parameters for a solver. We ran ParamILS with STP2 on a selection of problems both from the SMT-LIB library and from STP2 users. Three of the simplifications that we have discussed so far

Algorithm 3.7 Unsigned interval analysis with some operations omitted. Initially, the CALCULATE_INTERVAL procedure is called with the root expression, and empty *lower* and *upper* maps.

```
Require: $e$                                         // The current expression
Require: $lower$                     // Map from expressions to the lower bound
Require: $upper$                     // Map from expressions to the upper bound
 1: procedure CALCULATE_INTERVAL($e, lower, upper$)
 2:     if $e \in lower$ then
 3:         return        // Expression has already been evaluated. Only visit once.
 4:     end if
 5:     for each child $c$ of $e$ do CALCULATE_INTERVAL($c, lower, upper$)
 6:     end for
 7:     Create integer $u \leftarrow 2^{bitwidth(e)} - 1$          // assign 1 for formulae
 8:     Create integer $l \leftarrow 0$
 9:     if $e$ matches $true$ then
10:         $l \leftarrow u \leftarrow 1$
11:     else if $e$ matches $(bvnot\ t)$ then
12:         $l \leftarrow (bvnot\ upper(t))$
13:         $u \leftarrow (bvnot\ lower(t))$
14:     else if $e$ is a constant then
15:         $u \leftarrow l \leftarrow toInteger(e)$
16:     else if $e$ matches $(t_0 = t_1)$ then
17:         if $(lower(t_1) > upper(t_0)) \wedge (lower(t_0) > upper(t_1))$ then
18:             $l \leftarrow u \leftarrow 0$
19:         end if
20:     else if $e$ matches $(t_0 + t_1)$ then
21:         if $(upper(t_1) + upper(t_0))$ does not overflow then
22:             $l \leftarrow lower(t_0) + lower(t_1)$
23:             $u \leftarrow upper(t_0) + upper(t_1)$
24:         end if
25:     else if $e$ matches $ite(p, t_0, t_1)$ then
26:         $l \leftarrow min(lower(t_0), lower(t_1))$
27:         $u \leftarrow max(upper(t_0), upper(t_1))$
28:     end if
29:     if $l = u$ then
30:         Replace $e$ by constant $l$
31:     end if
32:     $upper(e) \leftarrow u$
33:     $lower(e) \leftarrow l$
34: end procedure
```

are disabled by default in STP2: AIG rewriting (section 3.12), AIG rewriting of the Boolean abstraction (section 3.13), and ITE Simplifications (section 3.14).

When we give the results in the next section (section 3.19), we show that enabling AIG rewriting solves extra problems. Because the test set that we use in our evaluation differs from the test set that we used to optimise the parameters, the best selections of simplifications to enable is different.

3.19 Evaluation

To build a test set, we took the SMT-LIB `QF_BV` benchmark set as of January 2012. Then we discarded the *asp* family of benchmarks which is large (29GB), and contains encodings of problems we are uninterested in, for example: towers of Hanoi, travelling salesperson, and Sudoku problems. We discarded the *mcm* family because it uses the define-fun syntax that STP2 cannot yet parse. We discarded the *bruttomesso-core* family because they contain no arithmetic. Next, we limited each family to 50 randomly chosen benchmarks that at least one of STP2 r1611 or Z3 3.2 [dMB08b], could not solve inside 1 second. We were left with 715 benchmarks in 31 families. Next we ran each problem using a memory limit of 3GB and a timeout of 500 seconds on a single core of an Intel E5507 Linux computer. This is the test set and configuration we use in the next chapter's evaluation, too (section 4.7).

There is a substantial variation in the solving time due to the *bruttomesso* families of hardware verification problems, so we report times for those families separately.

Table 3.2 compares STP2 r1654 with Z3 3.2, showing that STP2 is competitive with Z3. Table 3.3 shows the times for the *Bruttomesso* families; on these benchmarks, STP2 is not competitive with Z3. Overall STP2 performs well.

3.20 Relative Significance of Simplifications

To determine which of the simplifications we have presented are the most important, in this section we compare configurations of STP2 with individual simplifications disabled and enabled. The intention is to identify which simplifications are the most important.

In the evaluation, we use three different CNF simplifications. All of the CNF simplifications we use read and write the DIMACS CNF format, so can easily be used before SAT solving. The ability to simply use CNF simplification tools is an advantage of the eager approach. Integrating CNF simplification in the lazy SMT approach is much more time consuming. SatELite [EB05] converts a CNF into a simpler CNF by eliminating variables and subsumed clauses. We use the original

		STP2 with TM		STP2 r1654		STP2-4Simp		Z3 3.2	
Family	#	time	fail	time	fail	time	fail	time	fail
VS3	11	1	1/10	120	1/10	393	1/9	**610**	**7**
brummayerbiere	28	224	1/12	290	1/12	**670**	**1/11**	551	1/13
brummayerbiere2	50	1941	13	1903	1/12	**2897**	**9**	2127	1/29
brummayerbiere3	50	1524	24	1505	24	**1319**	**24**	1326	1/31
calypto	17	4	15	2	12	597	14	**969**	**11**
galois	3	**0**	**3**	**0**	**3**	**0**	**3**	**0**	**3**
gulwani-pldi08	3	58		**15**		27		17	
pipe	1	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
rubik	6	**114**		271		152	1	88	1
sage:app1	50	**52**		238		69		216	
sage:app12	14	**0**		**0**		**0**		**0**	
sage:app2	1	**0**		**0**		**0**		9	
sage:app7	6	**0**		**0**		**0**		10	
sage:app8	50	36		**19**		43		63	
sage:app9	50	38		**19**		44		60	
spear:cvs_v1.11.22	28	**42**		44		50		134	
spear:inn_v2.4.3	50	81		**46**		113		294	
spear:openldap_v2.3.35	5	0	5	**11**	**1**	475	2	0	5
spear:samba_v3.0.24	50	640		**119**		334		585	
spear:wget_v1.10.2	41	101		**83**		157		485	
spear:xinetd_v2.3.14	1	1		**0**		**0**		3	
spear:zebra_v0.95a	5	**3**		4		8		16	
stp	1	0	1/1	16		27		**9**	
stp_samples	22	2	2	3	2	2	2	**1**	**2**
tacas07	3	155	1	**256**		322		929	
uclid_contrib_smtcomp09	7	648		999		**645**		1598	
uclid:catchconv	50	0	1/50	**47**		92		137	
uum	7	35	6	31	6	17	6	**11**	**6**
wienand-cav2008:Booth	5	82	4	78	4	45	4	**30**	**4**
Sum	615	5791	147	6132	87	8511	86	10288	113
Time incl. penalty		79438s		49719s		51597s		66901s	

Table 3.2: Problems solved by: STP2 with all simplifications disabled except for TM, STP2 default configuration, 4Simp—a simple solver (section 3.22) , and the current version of SMT-COMP 2012 winner Z3 3.2. '#' is the number of problems in each family. 'time' is the time in seconds for successful instances. 'fail' is the number of failures. 1/19, means 19 failures in total, one of which exceeds the memory limit. 'Time incl. penalty' is the sum of the successful times plus 501 seconds penalty for each failure. The *bruttomesso* benchmarks are given in Table 3.3.

SatELite implementation, and the implementation of PrecoSAT 570. We also use blocked clause elimination [JBH10] from Precosat 570.

Table 3.4 gives the number of failures for STP2 configurations with single simplifications enabled or disabled. Not all of the simplifications that we have discussed

		STP2 with TM		STP2 r1654		4Simp		Z3 3.2	
Family	#	time	fail	time	fail	time	fail	time	fail
bruttomesso:lfsr	50	6861	5	7434	4	1760	1	**895**	
bruttomesso:simple_processor	50	3511	4	3576	5	1600	4	**1765**	**3**
Sum	100	10372	9	11010	9	3360	5	2661	3
Time incl. penalty		14881s		15519s		5865s		4164s	

Table 3.3: Bruttomesso families benchmarks solved by configurations. Headings are as per Table 3.2.

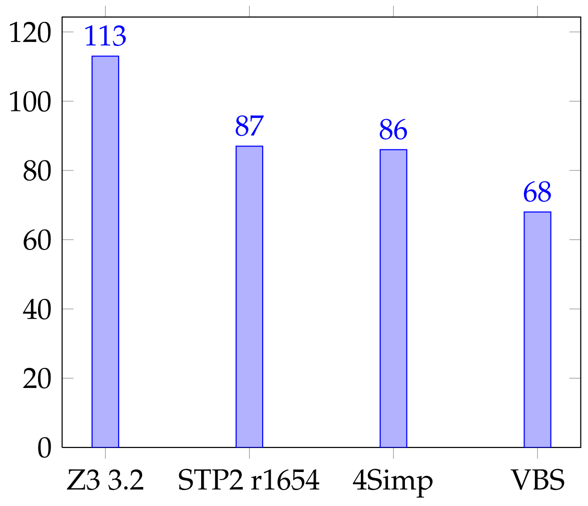


Figure 3.9: Number of times the memory limit of 3GB or the time limit of 500s was exceeded when running 615 SMT-LIB2 problems.

are enabled by default in STP2. The *Virtual Best Solver* (VBS) gives the number of problems that no variant could solve. The VBS effectively runs each configuration in parallel and gives the time when the first configuration solves a problem (if any). Note that some simplifications solve more problems than the default STP2 configuration, for instance, using AIG rewrites solves four extra problems. Table 3.6 gives the results just for the *Bruttomesso* families. Figure 3.9 shows the relative number of failures on the 615 test problems.

Table 3.4 contains comparisons between three approaches for generating CNF. The STP2 original CNF encoding, which was inherited from an earlier version of STP, does not use AIGs; it uses a custom propositional-to-CNF encoding scheme. It solved 45 fewer problems than the AIG and TM approach. We use the ABC tool version abc70930. The Tseitin CNF encoding, implemented by the ABC tool, uses AIGs, and converts via the Tseitin transformation to CNF. TM has the highest impact of any simplification. Note though that STP2 has various ways to encode the basic operations. For example, by selecting different configuration options, there are about 80 distinct ways to encode multiplication. We have used parameter optimisation to select good propositional encodings for bit-vector operations with the TM CNF translator. If parameter optimisation was reapplied with either the original or the Tseitin CNF transformation, then the difference would probably be reduced.

With *clause count comparison* disabled, 10 fewer problems are solved. So, the speculative transformations have made these 10 problems harder to solve. If the speculative transformations are disabled, 4 fewer instances are solved. So without the clause count comparison approach, the speculative transformations are harmful. Note that disabling the speculative transformations also disables the partial solver, which relies on the speculative transformations for correctness.

The virtual best solver answers 19 more problems than the default configuration. This demonstrates a challenge in building efficient bit-vector solvers: simplifications speed up some problems, but slow others down.

Table 3.5 shows the number of problems that were solved by particular configurations, that were not also solved by the default STP2 configuration. It shows that even though the original CNF encoding solved 45 fewer problems than the default STP2 configuration (Table 3.4), it solved one problem that the default STP2 configuration could not.

Table 3.5 also shows that disabling clause count comparison solves two problems that the default configuration does not. Disabling clause count comparison does not solve 12 problems the default configuration does, but solves two problems the default configuration fails upon. So the changes from speculative transformations are reverted for two problems, even though they made the problem easier to solve.

Table 3.6 gives the number of failures for different configurations when solving the *Bruttomesso* families. Unlike for the other benchmarks, where Glucose 2.0 solved the most problems, for the *Bruttomesso* families, Glucose 2.0 solves 25 fewer problems than Minisat 2.2.

Table 3.7 gives the measurements for STP2 with Glucose as a solver, and for STP2 and AIG rewriting. These configurations correspond to the best two configurations that we considered. Note that for the *spear* families, enabling AIG rewrites

# Failures	Configuration
132	Using STP original CNF encoding
103	Using ABC Tseitin CNF encoding
97	Disabling clause count comparison (section 3.7)
92	Enabling Precosat's Blocked clause elimination
91	Disabling speculative transformations and the partial solver
90	Disabling bit-blasting simplifications (section 3.8)
90	Disabling unconstrained simplification (section 3.15)
90	Disabling theory-level bit-propagation (chapter 4)
89	Disabling variable elimination (section 3.4)
89	Disabling creation-time simplifications (section 3.3)
88	Enabling AIG Boolean abstraction rewrite (section 3.13)
88	Disabling the pure literal rule (section 3.16)
88	Disabling the interval simplification (section 3.17)
87	Enabling the ITE simplifications (section 3.14)
87	Disabling the partial solver (section 3.5)
87	STP2 r1654 default configuration
86	Enabling Precosat's Satelite-style variable elimination
83	Enabling AIG rewriting (section 3.12)
81	Using SatELite CNF preprocessor and Glucose 2.0
68	Virtual Best Solver

Table 3.4: STP2 with simplifications enabled/disabled. The number of failures amongst 615 test problems excluding the *Bruttomesso* families is shown for each configuration. The fewer the failures the better.

# New problems solved	
1	Disabling speculative transformations and the partial solver
1	Enabling Precosat's Blocked clause elimination
1	Enabling AIG Boolean abstraction rewrite (section 3.13)
1	Disabling unconstrained simplification (section 3.15)
1	Using ABC Tseitin CNF encoding
1	Using STP original CNF encoding
1	Disabling the interval simplification (section 3.17)
1	Disabling theory-level bit-propagation (chapter 4)
2	Disabling clause count comparison (section 3.7)
5	Enabling Precosat's Satelite-style variable elimination
8	Enabling AIG rewriting (section 3.12)
10	Using SatELite CNF preprocessor and Glucose 2.0

Table 3.5: Problems solved by STP2 with specified configuration that were *not* also solved by the default STP2 configuration. This excludes the *Bruttomesso* families.

considerably slows down solving. These instances generally take a few seconds to solve, so the AIG rewriting is not justified for these. However, the *brummayerbiere3* family has far more problems solved with AIG rewriting enabled.

Failures	Configuration
45	Using ABC Tseitin CNF encoding
40	Using STP original CNF encoding
34	Using SatELite CNF preprocessor and Glucose 2.0
22	Enabling Precosat's Blocked clause elimination
10	Enabling AIG rewriting (section 3.12)
10	Disabling the pure literal rule (section 3.16)
10	Disabling the interval simplification (section 3.17)
10	Disabling theory-level bit-propagation (chapter 4)
10	Disabling the partial solver (section 3.5)
9	Disabling speculative transformations and the partial solver
9	Enabling AIG Boolean abstraction rewrite (section 3.13)
9	Disabling clause count comparison (section 3.7)
9	Disabling unconstrained simplification (section 3.15)
9	Enabling the ITE simplifications (section 3.14)
9	Disabling variable elimination (section 3.4)
9	Disabling creation-time simplifications (section 3.3)
9	STP2 r1654 default configuration
7	Disabling bit-blasting simplifications (section 3.8)
4	Enabling Precosat's Satelite-style variable elimination
1	Virtual Best Solver

Table 3.6: Number of failures for the 100 Bruttomesso test problems with various configurations of STP2.

3.21 A Comparison of the Tseitin and TM Encodings

Table 3.4 showed that disabling the TM simplification had the largest improvement of any simplification we investigated. In this section, we compare the CNF encoding of bit-vector operations via the Tseitin and TM encodings.

For several operations we measure how long unit propagation takes, and how many assignments are derived by unit propagation for each encoding. We encode each operation in an equality expression, which allows us measure how much unit propagation occurs. For instance, to measure the bit-vector exclusive-or's propagation, we encode $(v_0^{[64]} = (v_1^{[64]}\ bvxor\ v_2^{[64]}))$ to CNF. We randomly set all of the bits of (v_1, v_2) to one or zero with uniform probability, then calculate v_0. Next we delete 50% of the assignments. We apply unit propagation, but not search, to 100,000 such instances. After unit propagation completed, we counted the extra number of input and output bits that were assigned.

For some operations it is quick to compute the maximally precise (section 2.10) assignment. We discuss how we implement this in section 4.8. The results are

		STP2		STP2+AIG rewrites		STP2 + Glucose	
Family	#	time	fail	time	fail	time	fail
VS3	11	120	1/10	0	1/11	**186**	**1/9**
brummayerbiere	28	290	1/12	293	1/12	**380**	**1/11**
brummayerbiere2	50	1903	12	2113	3/12	**1670**	**11**
brummayerbiere3	50	1505	24	**1007**	**1/17**	1569	23
calypto	17	2	12	441	11	**855**	**9**
galois	3	**0**	**3**	**0**	**3**	**0**	**3**
gulwani-pldi08	3	15		18		**6**	
pipe	1	**0**	**1**	**0**	**1**	**0**	**1**
rubik	6	271		92		**64**	
sage:app1	50	**238**		400		247	
sage:app12	14	**0**		**0**		**0**	
sage:app2	1	**0**		**0**		**0**	
sage:app7	6	**0**		1		**0**	
sage:app8	50	**19**		48		22	
sage:app9	50	**19**		51		22	
spear:cvs_v1.11.22	28	44		131		**32**	
spear:inn_v2.4.3	50	**46**		659		91	
spear:openldap_v2.3.35	5	**11**	**1**	119	3	1394	2
spear:samba_v3.0.24	50	**119**		578		297	
spear:wget_v1.10.2	41	**83**		598		320	
spear:xinetd_v2.3.14	1	**0**		**0**		**0**	
spear:zebra_v0.95a	5	**4**		28		6	
stp	1	**16**		25		84	
stp_samples	22	**3**	**2**	4	2	**3**	**2**
tacas07	3	256		312		**150**	
uclid_contrib_smtcomp09	7	999		455	1	**447**	
uclid:catchconv	50	**47**		132		114	
uum	7	31	6	31	6	**9**	**6**
wienand-cav2008:Booth	5	78	4	90	4	**21**	**4**
Sum	615	6132	87	7639	83	8002	81
Time incl. penalty		49719s		49222s		48583s	

Table 3.7: Problems solved by various configurations. Columns are as per Table 3.2.

shown in Table 3.8. A higher percentage means more of the possible information was determined. The percentage given for the arithmetic shift is deceptively high because shifting random assignments is easy (we discuss this in section 4.10). The Plaisted and Greenbaum translation [PG86] is a more modern encoding, so it would make a better comparison for TM than comparing it to the Tseitin transformation. However, when measuring the SAT solving time of SMT-LIB bit-vector problems Jarvisalo et al. [JBH11] found only a small difference using each translation.

	TM Encoding				Tseitin Encoding			
operation	clauses	time	extra	%	clauses	time	extra	%
signed ≥	693	0.95	17296	79	1340	1.87	15932	73
unsigned less than	681	0.97	17337	80	1325	1.90	15903	73
equal	310	0.43	49994	100	767	0.82	49826	100
bit-vector xor	384	0.98	2398227	100	704	1.45	2400319	100
bit-vector or	320	0.87	2799457	100	320	0.91	2797177	100
bit-vector and	320	0.75	2800824	100	320	0.70	2799984	100
arithmetic shift	2114	1.44	3249159	100	4289	2.85	3249174	100
addition	1011	1.78	1136975	67	2204	3.28	1138400	67
subtraction	1011	1.76	1139723	67	2204	2.92	1140825	67
multiplication	34350	20.45	148453	–	71504	81.64	148273	–
unsigned division	63738	117.99	3038078	–	166091	447.68	3038667	–
unsigned remainder	64074	124.02	757930	–	167429	456.71	719392	–
signed division	65624	62.05	1163098	–	170048	248.40	1065681	–
signed remainder	65761	61.96	211400	–	171377	232.60	164152	–

Table 3.8: A comparison of operations encoded via ABC's Tseitin and TM transformation. 100,000 iterations on a single core of an Intel Q8400 Linux computer were run. 50% of variables have a known assignment initially. 'clauses' is how many clauses the encoding contains (including an extra equality). 'time' is the sum of the time in seconds to perform unit propagations. 'extra' is the total number of extra assignment unit propagation determined. '%' is the percentage of the maximum possible number of assignments that were discovered. That is, the percentage of assignments discovered versus the maximally precise propagator. The % is not shown for some operators for which it is too expensive to calculate the maximally precise result.

In this section we do not measure the percentage of unsatisfiable assignments that are detected via unit propagation. This is another useful measure of an encoding's propagation strength.

The results show why the TM encoding is better than the Tseitin encoding. The TM encoding of the operations never has more clauses than the Tseitin encoding. Applying unit propagation to the TM encoding is sometimes much faster. For instance, unit propagation on the multiplication TM encoding is four times faster than applying it to the Tseitin encoding. The result of applying unit propagation to the TM encoding does not assign fewer variables. The TM encoding has fewer clauses, unit propagation completes faster on it, and more variables' assignments are deduced for the comparison operations.

3.22 A Simple Fast Solver (4Simp)

We have already turned off single simplifications and measured the effect of this (section 3.20). In this section, we complement the prior sections by measuring STP2 with just a few simplifications enabled. We answer the question: "Which simplifications make a simple, fast bit-vector solver?". We show just that a few simplifications are the most important.

We start with STP2 r1654 with all simplifications disabled. Based on the prior results, TM was the single most important simplification, so we enable just that. The workflow of STP2 in this configuration is simple. Problems are parsed, structurally hashed, bit-blasted to AIGs, then encoded via TM to CNF. The results are shown in Table 3.2; with this configuration 147 benchmarks failed.

By trial and error we determined that turning on variable elimination and creation-time simplifications solved many of the *uclid:catchconv* family—all 50 of which failed with just TM enabled. Because it is easy to run the SatELite-style simplification, and the prior sections showed it helped, we enabled that too. We call this solver *4Simp*; it is STP2 with only creation-time simplifications, variable elimination, Precosat's SatELite-style simplification, and TM. It solves one more problem than the default STP2 configuration, and 27 more than Z3 3.2. Enabling some other simplifications gives even better performance, but our intention is to show that a few simplifications are enough to build a competitive bit-vector solver. Of course, on different tests, different simplifications might help.

STP2 performs worse than 4Simp on the problems we have evaluated with. This is because STP2 has been tuned to solve problems from a different test set—those provided over time by users of STP. For the problems we evaluated with in this section, the 4Simp solver is better choice of simplifications.

We compared 4Simp using the variable elimination algorithm that we described in section 3.4 versus 4Simp with a simpler variable elimination algorithm which only eliminates variables which are asserted to equal terms at the top level. For the SMT-COMP 2012 `QF_BV` division problems, the simpler variable elimination algorithm solved 189 problem, while the algorithm we describe solved 188.

3.23 Related Work

Bit-vector solvers have a rich history inspired initially by problems from electronics design automation. At SMT-COMP 2011, in the `QF_BV` division, the five top placed solvers were bit-blasting based solvers. Eager SAT based approaches are currently dominant at solving bit-vector problems derived from software. In this section, we focus mostly on differences between eager bit-vector solvers. In subsection 3.23.7, we discuss alternative approaches.

Tracy Larrabee [Lar90] bit-blasted circuits and used a SAT solver to discover test cases for hardware circuits. However, BDDs, popularised by Bryant [Bry86], dominated hardware verification problems until Biere et al. [BCCZ99] showed good results with a SAT approach.

Many variants of BDDs were tried, like *BMD$_{()}$ [Ard96] which have linear rather than BDDs' exponential memory growth as the bit-width of multiplication grows. The size of BDDs, and so the memory used, depends on its variable ordering, if a good variable ordering can be found, then for some problems BDDs are still faster than SAT [SD11].

Many theorem provers, such as ACL2, have bit-vector libraries [Rus99] to allow bit-vector reasoning inside the prover. To increase automation (theorem provers often require human intervention to direct the search for a proof), libraries have been implemented which can discharge theorems by bit-blasting to SAT [Fox11]. Proofs of unsatisfiability which are generated by some bit-vector solver, for instance Z3 [dMB08b], but not by STP2, can be automatically checked by the theorem prover to increase the confidence the result is correct [BFSW11]. Böhme et al. [BFSW11] compare a bit-blasting algorithm [Fox11] for HOL4 versus that of Z3, and find that Z3 was substantially faster at solving `QF_ABV` problems.

In another context, Huang [Hua08] has used bit-blasting to solve finite domain constraint programming problems.

Equivalence checking problems ask whether, for the same inputs, the output of two circuits can differ. Bit-blasting has been used in part to solve equivalence checking problems in hardware [KJJP09], and software [Smi11].

Little work has focused on solving problems in the quantified theory of bit-vectors. John and Chakraborty [JC11] perform quantifier elimination before solving with STP. Wintersteiger et al. [WHdM10] apply a simplification phase to quantified problems before instantiating quantifiers.

Fragments of QF_BV are decidable in polynomial time. For instance, Cyrluk et al. [CMR97] give an algorithm for solving problems with bit-vector variables, bit-vector constants, concatenation, extraction and equality in polynomial time in the number of equalities.

Our results (section 3.19) showed that the *virtual best solver* solved 27 more problems than the default STP2 configuration. Adjusting the solver approach to fit with particular problems is clearly advantageous. De Moura and Passmore (unpublished [dMP12]) refer to the "strategy challenge" and advocate giving users the ability to exert control over the heuristics used to solve problems, that is, giving informed users the ability to specify the simplifications and solving techniques that are applied to a particular problem.

Portfolio solving runs different, or differently configured, solvers in parallel to solve a problem. The approach is successful for SAT solving [XHHLB08]. The difference between the virtual best solver, and the default configuration of STP2 show that a similar approach might be useful for bit-vector solving.

3.23.1 Spear

Domagoj Babić [Bab08] describes the Spear solver which won the SMT-COMP 2007 QF_BV division. Spear is an eager bit-vector solver. It rewrites the input expression, then encodes to CNF using encodings of operations that are optimised to have the few logical operations. Spear then simplifies the CNF before SAT solving.

A major novelty of Spear is the automatic parameter tuning of its SAT solver. We also used the ParamILS tool as described in section 3.18. Automatically tuning the SAT parameters reduced the average runtime of Spear on some problems from 780 seconds to 1.5 seconds.

Other potentially significant differences to STP2 are: the use of Guild divisor—a type of division circuit, and the conversion of division by constants into multiplications.

3.23.2 MathSAT

Roberto Bruttomesso [Bru08] described a bit-vector solver that is incorporated into the lazy MathSAT solver. MathSAT calculates the boolean abstraction (introduced in section 3.13) of bit-vector problems, and then uses a layered approach to solving. Each potentially spurious model is checked by an equality with uninterpreted functions solver before bit-vector solving occurs.

MathSAT uses an uninterpreted function solver to look for obviously inconsistent sets of assignments. For example, $((1 \times 2) = 4) \wedge ((1 \times 2) = 3)$ is inconsistent because of functional congruence, irrespective of the interpretation of the multiplication operation.

Bruttomesso gives rules for the unconstrained simplification (section 3.15) that we followed with STP2.

3.23.3 UCLID

UCLID as described in Bryant et al. [BKO+07] uses under- and over-approximation of bit-vector problems. The problem is first under-approximated (i.e. replaced by one that entails it) and solved, and if the under-approximation is satisfiable, the procedure has completed. If it is unsatisfiable, the unsatisfiable core is examined and used to produce an over-approximation of the input.

To produce the under-approximation, the topmost bits of variables are constrained to all be equal. For instance, given a variable $\langle v[3], v[2], v[1], v[0]\rangle$, $v[3]$ through $v[1]$ might be constrained to all have the same value. Each time an under-approximation is produced, the number of top-most bits thus constrained is reduced.

The over-approximation is produced by replacing each Boolean node with a fresh unconstrained variable, whenever the node does not appear in the under-approximation's unsatisfiable core.

During its abstraction phase, UCLID replaces hard operations with partial implementations. For instance, the multiplication of more than 4 bits is replaced by the partial implementation: $ite(x = 0 \vee y = 0, 0, ite(x = 1, y, ite(y = 1, x, mul(x, y))))$, where mul is a uninterpreted function symbol.

3.23.4 Boolector

Robert Brummayer [Bru09] describes the open-source Boolector eager bit-vector and array solver.

Brummayer and Biere [BB09] describe an under-approximation technique to speed up solving unsatisfiable formulae. This is a more sophisticated implementation of the idea of Bryant et al. [BKO+07]. Extra constraints are encoded with the problem so that it is over-constrained. The bottom n bits of a bit-vector have no additional constraints, while the topmost m bits are constrained. The original formula is translated to CNF, with extra constraints.

For instance, to constrain the topmost two bits of $t^{[8]}$ to the sign extension, a new variable e is added to the CNF. Then the additional clauses $(e \rightarrow (t[7] = t[5])) \wedge (e \rightarrow (v[6] = v[5]))$ are asserted to the SAT solver, and e is assumed. If the SAT solver reports that the problem is satisfiable, then the process is finished. However, if it is unsatisfiable, the actual models might have been erroneously removed, so it is necessary to assert $\neg e$. Assumptions allow constraints to be removed from the SAT solver while keeping some of the learnt conflict clauses. To avoid too many refinement loops, the effective bit-width n is usually doubled with each refinement. If the SAT solver yields "unsatisfiable", and none of the assumptions were used to derive the contradiction, then it is no longer necessary to search.

3.23.5 Z3

The Z3 solver [dMB11], which recently had its source code published, is the most widely used and capable SMT solver. Z3 supports the combination of many different theories, as well as features that STP2 does not implement, such as producing reasons for unsatisfiability and interpolants. There is limited published information about Z3's bit-vector implementation. In a mailing list, de Moura [dM11] describes that `QF_BV` solving in Z3 version 3 is based on preprocessing, then bit-blasting.

3.23.6 Beaver

Beaver [LS10] is an eager `QF_BV` solver. It performs rewrites followed by conversions to AIGs then via the ABC tool's TM to CNF. Beaver pre-computes AIG templates for expensive operations (multiplication, addition, division and remainder). When these operations are bit-blasted to AIGs, these pre-simplified templates are instantiated. For instance, at design-time multiplication is encoded from Verilog into an AIG, then the ABC package's AIG optimisations are used to simplify the encoding. Performing this simplification at runtime would be too expensive. The technical report [LS10] compares the solving time using these optimised versus unoptimised templates, showing that the optimised templates are helpful. The optimised templates have two advantages: first, bit-blasting time is lowered because the templates are cheap to instantiate. Second, the templates can be carefully optimised off-line. We have not measured if this is also useful for STP2.

Another novelty of Beaver is that it replaces modulus, remainder and division operations by multiplication. Excluding consideration of division by zero, $(a \div_u b)$ is replaced by q, with the additional constraint $a = qb + r \wedge r < b$, constrained at the top level. The addition and multiplication that are introduced are specially constrained to avoid overflow. The authors justify the rewriting of division, modulus and remainder to multiplication as being useful because division generates larger circuits as compared to multiplication. In Table 3.8, we showed that the multiplication operation's encoding has about half the number of clauses compared to division. We experimented with converting unsigned division to multiplication, but did not get a speedup.

When solving SMT-LIB problems, of the SAT solvers Limaye and Seshia experimented with, they found the non-clausal SAT Solver NFLSAT [JC09], which can read AIG input, the fastest.

Jha et al. [JLS09] also compare the ABC tool's implementation of TM versus the Tseitin encoding. The scatter plots they give show that TM is faster, but they do not quantify the difference. They found TM to significantly speed up the *spear* families. STP2 r1654 with TM solves 180 spear problems in 460 seconds, whereas with Tseitin it takes 310 seconds. We saw the largest difference in the *Bruttomesso* families, where the use of TM led to solving 36 more problems.

3.23.7 Other Approaches

We now discuss a few approaches other than the eager encoding to solve bit-vector problems.

Lazy SMT. The traditional SMT approach is the lazy approach [Seb07]. It follows the ideas of the Nelson-Oppen combination method, and abstracts the problem into a Boolean abstraction, on which the SAT solver produces candidate assignments that theory solvers check for consistency. The lazy SMT approach is good for combining theories, and for dealing with problems that have a large or infinite eager CNF encoding.

Integer Linear Programming. Zeng et al. [ZKC01] converts bit-vector problems to integer linear problems.

Example 3.26

To linearise the logical *and* operation, where $(a = (b \wedge c))$, Zeng et al. [ZKC01] encode using integer variables that are either 0 or 1, and then assert: $a_i \leq b_i$, $a_i \leq c_i$, and $a_i \geq b_i + c_i - 1$. ■

Achterberg [Ach07] linearises bit-vector problems and solves them using a standard linear solver. He does not linearise all bit-vector operations, reporting that linearisation of the shift-left constraint on a 64-bit input requires 30944 inequalities, and 20929 new variables; too many to be practical. Bruttomesso [Bru08] also investigated linearisation for solving bit-vector problems.

Propagators. Bardin et al. [BHP10] built propagators for two domains for each bit-vector operation. One domain is for constant bits, the same domain we investigate in chapter 4. The other is sets of unsigned intervals. Information is kept updated between the two domains. Using a worklist, all the propagators are run until a fixed point is reached, and then search occurs. An advantage of their approach is that the encoding size does not increase quadratically with the bit-width of the expressions. Their solver slows down a little bit as the size of problems is changed from 64 to 512 bits, but less drastically than the bit-blasting solvers slow down. Their propagators are not able to explain, in the sense of lazy clause generation [OSC09], why a conflict occurred. So, unlike SAT based approaches, there is no conflict driven clause learning to prune the search space.

3.23.8 Bit-Width Reduction

Bit-width reduction produces an equisatisfiable problem where the expressions have fewer bits.

These reductions can be performed in the presence of bitwise operations like "and", "or", and exclusive-or which operate uniformly on all the bits. That is, each bit i of the output is a function just of the bit i's of the inputs. Care needs to be taken to ensure that the reduced bit-width is large enough to allow equations to be transitively equals / not equals. Johannsen and Drechsler [JD01] reduce the encoding of a hardware verification problem by 70% by applying the simplification upfront. This is similar to the decision procedure of Cyrluk et al. [CMR97].

Example 3.27

Consider the expression $(y^{[32]}[15,0] = y^{[32]}[31,16])$, where these are the only occurrences of y. An equisatisfiable expression with a fresh 2-bit variable is: $(v^{[2]}[1,1] = v^{[2]}[0,0])$. To convert a model of $v^{[2]}$ into a model of $y^{[32]}$, let $y^{[32]} = (0^{[15]} :: (y^{[2]}[0,0] :: (0^{[15]} :: y^{[2]}[1:1])))$. ■

We do not implement this simplification because the arithmetic operations commonly contained in software verification problems, such as addition and multiplication, do not operate uniformly on the bit-vector operands.

3.23.9 Peephole Optimisation

A peephole optimiser is a rewrite system in a compiler that replaces sequences of instructions with other equivalent, but better sequences of instructions. It is called a peephole optimisation because the replacement is made locally, while looking at a small piece of the program.

Sorav Bansal [Ban08] automatically derives peephole rules. The idea is to enumerate instruction sequences, then run those sequences on a few inputs. The result of the instructions is used to build a hash value, and the instruction sequence is stored in a hash table. When two instruction sequences with the same fingerprint are found, both sequences are run on extra inputs. If the output of each sequence is the same, then the sequences are encoded to SAT and checked for equivalence.

If they are equivalent, then a rewrite rule is produced which replaces the inferior sequence with the better one. We applied a similar idea when we generated equivalences (subsection 3.11.1).

The principal differences between finding rewrite rules for peephole and bit-vectors are: the rules for bit-vectors should apply to all bit-widths; bit-vectors have a single output whereas machine instructions change processor flags, registers and memory; and instructions have irrelevant instructions intermixed, that is, the data dependency is not clear.

3.24 Conclusion

Research into `QF_BV` solvers aims to produce correct and faster solvers. We have described the architecture and simplifications that we developed to make STP2 and 4Simp, competitive modern bit-vector solvers.

This chapter contains descriptions of some novel approaches. In particular:

- The variable elimination algorithm (section 3.4), is a principled approach to isolating variables.
- The bit-blasting equivalence checking (section 3.8), transfers information derived by the AIGs back to the bit-vector theory-level.
- The approach to discovering equivalences (subsection 3.11.1), gives a way for authors of bit-vector solvers to discover equivalences that might be useful.

We showed how to automatically generate bit-vector equivalences by comparing bit-vector terms on a range of bit-widths. We found it most useful to use the equivalences that were discovered to check whether there were extra rules that we could include into STP2 and 4Simp. We found that too few of the rules matched larger instances, making it impractical to apply them.

We have explored the effect of simplifications on solving bit-vector problems. Of the simplifications we discussed, on the benchmarks we examined, the TM approach ([EMS07]) made the most dramatic improvement. We showed that applying it, along with creation-time simplifications, variable elimination, and SatELite preprocessing was enough to achieve a simple and competitive bit-vector solver.

In the next chapter we examine another transformation.

4
Theory-Level Bit Propagation

4.1 Introduction

IN this chapter we investigate whether it is useful for STP2 to have a simplification phase which calculates, at the theory-level, which bits must be true or false. STP2 was described in chapter 3.

We consider the case where argument or result values are partially known, that is, some bit values are known. Reasoning at this level has the potential to expose bit relationships that will be much harder to identify after the high level structure has been "lost in translation", that is, after a problem has been encoded in CNF. Our aim here is to explore whether the approach scales well enough to be useful for larger realistic examples.

It is important to understand that bit propagation, as we use the term, deals with multi-way information flow. In compiler theory, "constant propagation" is a "forwards" analysis, in which values of expressions may be deduced from the values of sub-expressions, and bit-vector solvers often incorporate this. With bit propagation we aim not only to deduce bit values of composite expressions from their sub-expressions' known bit values, but also, simultaneously, to deduce bit values of the sub-expressions from what is known about the composite expression's bit values.

Intuitively, for multiplication and allied operations, the relationships amongst result and argument bits are highly complex. However, for the most and least significant bit positions, important relationships can be extracted with relatively little effort, and this is often sufficient to enable constraint simplification or improved implicativity of generated clauses.

Our idea is as follows. Before encoding into CNF (as presented to a SAT solver), we apply inexpensive propagators that deduce some of the bit values that the input, output and intermediate values must take for the generated clauses to be satisfiable. We use the information that these propagators establish to simplify expressions before encoding, that is, to replace sub-expressions and variables with known values.

We aim to simplify problems expressed in the QF_BV language, a quantifier free theory of fixed-width bit-vectors. Let us briefly recall (from section 2.3): We use $t^{[n]}$ to represent a bit-vector expression t of bit-width n, where $n > 0$. We use square brackets for the extract operation, which extracts a single bit or a sequence of bits. $t[0]$ is the least significant (or rightmost) bit. Unsigned arithmetic operations interpret the bit-vector as the integer $\sum_{i=0}^{n-1} 2^i \times t[i]$. We indicate bit-vector constants as strings of 0s and 1s. For example, the unsigned integer corresponding to $(110)_2$ is 6. Multiplication and addition are performed modulo 2^n, so the result may overflow—which significantly complicates the analysis. Unsigned division performs truncating integer division, which never overflows. Signed remainder gives the remainder of signed division with rounding toward zero. Signed modulus gives the remainder of signed division with rounding toward negative infinity.

For each of the operations in the QF_BV language, we have built propagators that deduce bits' values from operations' inputs and output. We use these deduced bit values to simplify the problem before it is encoded to CNF, while the problem is still at the theory-level. This can identify some simplifications that are harder to find in a CNF encoding.

Another advantage of implementing propagators rather than bit-blasting to CNF is that propagators use less memory per operation. The CNF encoding of some operations, like signed division, is large. For instance, STP2 encodes a 64-bit signed division as about 65,000 clauses (Table 3.8); for each such operation STP2 uses 20MB of memory, greatly limiting the number of operations that STP2 can handle. A 32-bit signed division is encoded as 17,500 clauses, and 128-bit signed division encoded as 262,000 clauses. The number of clauses and the memory used grows

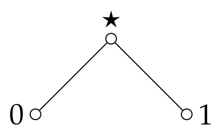


Figure 4.1: The ternary domain (**3**)

$\wedge$	0	1	★
0	0	0	0
1	0	1	★
★	0	★	★

$\vee$	0	1	★
0	0	1	★
1	1	1	1
★	★	1	★

$\oplus$	0	1	★
0	0	1	★
1	1	0	★
★	★	★	★

$\neg$	
0	1
1	0
★	★

Figure 4.2: Truth tables in a three valued logic for *and*, *or*, *xor*, and negation, as given by Kleene's strong three-valued logic K_3

roughly quadratically with the bit-width. The memory use of the propagators we describe in this chapter grows more slowly.

To reason about the values that separate bits may take, it is useful to introduce three-valued logic. Let $\mathbf{2} = \{0, 1\}$ be the set of classical truth values. Figure 4.1 shows a Hasse diagram for the set of *ternary* truth values $\mathbf{3} = \{0, 1, \star\}$. As can be seen, the ordering $\leq$ of these values is defined by $v \leq v'$ iff $(v = v') \vee (v' = \star)$. That is, $\leq$ is an ordering on *information content*: 0 and 1 are incomparable, but equally informative elements, whereas $\star$ represents absence of information. We can give the semantics of elements of **3** with a function $\gamma :: \mathbf{3} \to \mathscr{P}(\mathbf{2})$ specifying the set of truth values each element of **3** corresponds to: $\gamma(0) = \{0\}, \gamma(1) = \{1\}, \gamma(\star) = \{0, 1\}$. Propositional logic's strongest monotone extension to **3** is known as Kleene's (strong) 3-valued logic and is used extensively in the fields of program transformation and verification to reason about partial functions. Truth tables for Kleene's logic, often denoted K_3, are given in Figure 4.2.

In the next section we define (ternary) truth assignments, but for now we take the liberty of using ternary bit vectors with the "obvious" meaning. For example, we use $\langle 00 \star 1 \rangle$ to denote a set of (classical) bit-vectors of length 4, containing two vectors $\{(0001)_2, (0011)_2\}$.

The input to our analysis is a formula, a well typed `QF_BV` expression of propositional type. The satisfiability problem is to find assignments to the variables that make the expression true. If there are no possible assignments, the formula is unsatisfiable, or equivalent to false. Before analysis, syntactically identical sub-expressions are shared (structurally hashed), giving a rooted DAG, with a propositional root node. When we perform an analysis using **3**, we calculate the sets of possible values at every node. We consider the value of the expression to come out the top (the root), and variables and constants to be the leaves at the bottom.

We assume the presence of a map M which aids bit-blasting. M maps `QF_BV` syntactic expressions to tuples of variables (ranging over $\mathbf{3}$). M maps a propositional expression to a single variable, and a bit-vector expression of bit-width n to a vector of n variables. A 3-valued assignment μ maps variables to elements of $\mathbf{3}$. Propagation works by updating this assignment μ.

In our analysis we begin by associating the output of each `QF_BV` node in the input expression with a vector of fresh variables, then set each variable to $\star$. Logical operations are associated with a vector of size one, bit-vector operations with fresh vectors of the appropriate width (see example below).

Because equalities are constraints, they may evaluate to 1 or 0. When it is not clear from context we indicate that an equality must be true with a superscript $=^t$, similarly when it must be false as $=^f$.

Occasionally we shall silently assume the presence of M and μ. For example, for brevity we may write $\langle 10\rangle + \langle 01\rangle =^t \langle\star\star\rangle$ for the the equation $t_0^{[2]} + t_1^{[2]} = t_2^{[2]}$, in the context of M and μ defined as

$$
\begin{aligned}
&M[t_0] = \langle o_0, o_1\rangle\\
&M[t_1] = \langle o_2, o_3\rangle\\
&M[t_0 + t_1] = \langle o_4, o_5\rangle\\
&M[t_2] = \langle o_6, o_7\rangle\\
&M[t_0 + t_1 = t_2] = \langle o_8\rangle\\
&\mu = \{o_0 \mapsto 1, o_1 \mapsto 0, o_2 \mapsto 0, o_3 \mapsto 1, o_4 \mapsto 1, o_5 \mapsto 1, o_6 \mapsto \star, o_7 \mapsto \star, o_8 \mapsto 1\}
\end{aligned}
$$

As an example of performing bit propagation, consider the expression $((b_0 \vee b_1) \wedge (v_0^{[3]} <_u (4^{[3]} \times v_1^{[3]})))$.

We first create a partial assignment of each node to an appropriately sized fresh vector $\langle o_i, \ldots, o_j\rangle$ of variables. We map constants directly to vectors of 1 or 0 in $\mathbf{3}$:

$$
\begin{aligned}
M((b_0 \vee b_1) \wedge (v_0^{[3]} <_u (4^{[3]} \times v_1^{[3]}))) &= \langle o_0\rangle\\
M(b_0 \vee b_1) &= \langle o_1\rangle\\
M(v_0^{[3]} <_u (4^{[3]} \times v_1^{[3]})) &= \langle o_2\rangle
\end{aligned}
$$

$$M(4^{[3]} \times v_1^{[3]}) = \langle o_5, o_4, o_3 \rangle$$
$$M(b_1) = \langle o_9 \rangle$$
$$M(b_0) = \langle o_{10} \rangle$$
$$M(4^{[3]}) = \langle 100 \rangle$$
$$M(v_1^{[3]}) = \langle o_{11}, o_{12}, o_{13} \rangle$$
$$M(v_0^{[3]}) = \langle o_{14}, o_{15}, o_{16} \rangle$$

Next we propagate bits in any expressions that have operands or results that are known bits.

- $((4^{[3]} \times v_1^{[3]})) \equiv \langle o_5, o_4, o_3 \rangle$ sets $o_3 \leftarrow 0$ and $o_4 \leftarrow 0$.

Then, the value of the expression o_0 is set to 1 (true) in the partial assignment: $o_0 \leftarrow 1$. Next propagators are applied until a global fixed point is reached:

- $o_1 \wedge o_2 \equiv o_0$, where $o_0 = 1$, sets $o_1 \leftarrow 1$ and $o_2 \leftarrow 1$.
- $(M(v_0^{[3]}) < \langle o_5, 0, 0 \rangle) \equiv \langle 1 \rangle$ sets $o_{14} \leftarrow 0$ and $o_5 \leftarrow 1$.
- $((M(4^{[3]}) \times M(v_1^{[3]}))) \equiv \langle 1, 0, 0 \rangle$ sets $o_{13} \leftarrow 1$.

Note that constraint propagation has deduced the value for one bit in each of $v_0^{[3]}$ and $v_1^{[3]}$, as well as some intermediate values. We can now conjoin these values with the CNF clauses that are sent to the SAT solver. The analysis we perform is not complete, but that is of little concern, since the SAT solver provides completeness.

Constant propagation is commonly implemented in bit-vector SMT solvers [LS10, BH08] and it has been studied in other contexts (see section 4.11). STP2, used in our evaluation later, performs constant propagation. There are cases where theory-level bit propagation, as introduced in this chapter, can surpass ordinary constant propagation. For example, bit propagation can determine that the formula $(1^{[1]} :: x^{[3]}) = (0^{[1]} :: y^{[3]})$ where "(::)" is concatenation, must evaluate to 0, something constant value propagation cannot.

4.2 Preliminaries

We now introduce the set of *partial truth assignments* as an ordered structure, related to *sets* of classical truth assignments. Let $\mathcal{A}_2 = Var \to \mathbf{2}$ be the set of 2-valued (classical) truth assignments and let $\mathcal{A}_3 = (Var \to \mathbf{3}) \cup \{\bot\}$ be the set of 3-valued truth assignments, extended with a special element $\bot$. The ordering $\leq$ on $\mathcal{A}_3$ is defined as follows: $\mu \leq \mu'$ iff $\mu = \bot \vee \mu(v) \leq \mu'(v)$ for all $v \in Var$.

We also define the set of concrete (2-valued) and abstract (3-valued) bit vectors, parameterised by bit-width, as

$$\mathcal{V}_2^{[n]} = \mathbf{2}^n$$
$$\mathcal{V}_3^{[n]} = \mathbf{3}^n$$

where $\mathbf{2}^n$ and $\mathbf{3}^n$ denote the sets of n-tuples of elements of $\mathbf{2}$ and $\mathbf{3}$, respectively. We lift our semantic function for $\mathbf{3}$ to $\gamma :: \mathcal{V}_3^{[n]} \to \mathscr{P}(\mathcal{V}_2^{[n]})$ in the obvious way:

$$\gamma((x_1, \ldots x_n)_2) = \gamma(x_1) \times \cdots \times \gamma(x_n)$$

For example, $\gamma((\star 0 \star 1)_2) = \{0,1\} \times \{0\} \times \{0,1\} \times \{1\} = \{(0001)_2, (0011)_2, (1001)_2, (1011)_2\}$.

We need to reason about relations over bit vectors (we are mostly interested in arithmetic functions, but since our propagators can also propagate information from outputs to inputs, it is most convenient to view n-ary functions as $(n+1)$-ary relations). An n-ary relation on m-bit vectors is a set of n-tuples of m-bit vectors listing the valid combinations of inputs and outputs. For example, 1-bit addition (as well as exclusive or) is captured by the relation

$$\{\langle (0)_2, (0)_2, (0)_2 \rangle, \langle (0)_2, (1)_2, (1)_2 \rangle, \langle (1)_2, (0)_2, (1)_2 \rangle, \langle (1)_2, (1)_2, (0)_2 \rangle\}$$

Thus it is convenient to define the sets of concrete and abstract n-tuples of m-bit vectors, ordered component-wise:

$$\mathcal{T}_2^{[m]\langle n \rangle} = (\mathcal{V}_2^{[m]})^n$$
$$\mathcal{T}_3^{[m]\langle n \rangle} = \bot \cup (\mathcal{V}_3^{[m]})^n$$

Then a concrete relation is a set of tuples:

$$\mathcal{R}_2^{[m]\langle n\rangle} = \mathscr{P}(\mathcal{T}_2^{[m]\langle n\rangle})$$

Note that we introduce a bottom element $\perp$ to our set of abstract values to serve as the abstraction for an empty set of concrete tuples. Also note that $\mathcal{V}_3^{[m]}$ (and hence $(\mathcal{V}_3^{[m]})^n$) is a join-semilattice, which suffices to give $\mathcal{T}_3^{[m]\langle n\rangle}$ the structure of a lattice.

We define the semantics of an abstract tuple by lifting the γ function to tuples much as we lifted it to bit vectors. We also define an abstraction function α to give the best abstraction for a set of concrete tuples.

$$\begin{aligned}
&\gamma :: \mathcal{T}_3^{[m]\langle n\rangle} \to \mathscr{P}(\mathcal{T}_2^{[m]\langle n\rangle})\\
&\alpha :: \mathscr{P}(\mathcal{T}_2^{[m]\langle n\rangle}) \to \mathcal{T}_3^{[m]\langle n\rangle}\\
&\gamma(t) = \begin{cases} \varnothing & \text{if } t = \perp \\ \gamma(x_1) \times \cdots \times \gamma(x_n) & \text{if } t = \langle x_1, \ldots x_n\rangle \end{cases}\\
&\alpha(S) = \bigsqcup S
\end{aligned}$$

Here $\bigsqcup$ is the least upper bound operator on $\mathcal{T}_3^{[m]\langle n\rangle}$. Note that $\mathbf{2} \subseteq \mathbf{3}$, so if $\langle x_1, \ldots, x_n\rangle \in \mathcal{T}_2^{[m]\langle n\rangle}$ then $\langle x_1, \ldots, x_n\rangle \in \mathcal{T}_3^{[m]\langle n\rangle}$.

A propagator uses what we know about the behaviour of a function to derive extra information about the function's inputs and outputs from the information supplied. The input to a propagator is a single tuple of abstract values, and the output is the same as the input tuple, but perhaps strengthened. Thus a propagator for an $(n-1)$-ary function on m-bit integers has type

$$\mathcal{P}_3^{[m]\langle n\rangle} = \mathcal{T}_3^{[m]\langle n\rangle} \to \mathcal{T}_3^{[m]\langle n\rangle}$$

and is ordered point-wise. Note that a propagator can only ever strengthen its input, so it must be reductive (section 2.10). It also makes no sense for a propagator ever to produce a weaker output from a stronger input, so it is required to be monotone.

Given a concrete relation R, we can define the *optimal* propagator P_R formally, like so:

$$P_R(t) = \alpha(R \cap \gamma(t))$$

Informally, this says that P_R is maximally precise. A propagator p for an n-ary relation R is *sound* iff $P_R \le p$.

All the propagators that we build are sound. Apart from the cases of division, remainder and multiplication, we have not found instances where they are not optimal, although we have not proved optimality formally, for most operations. An efficient optimal propagator for multiplication would solve cryptographically important factorisation problems, so the likelihood of discovering such a propagator is low. In section 4.8 we describe how we have tested our propagators.

We have implemented propagation for all `QF_BV` operations; in this chapter we focus on the more interesting propagators, namely those for bit-wise and, addition, multiplication, and unsigned division.

The propagators we implement allow the result of an expression to partially determine the inputs. Consider $((t_0^{[2]}\ \mathit{bvand}\ t_1^{[2]}) =^t t_2^{[2]})$, where $\mu = \{t_0 = \langle\star\star\rangle,$ $t_1 = \langle\star 0\rangle,\ t_2 = \langle 1\star\rangle\}$. Our propagators use the inputs and outputs to refine the other values, giving additional information about inputs t_0 and t_1: $\mu = \{t_0 = \langle 1\star\rangle,$ $t_1 = \langle 10\rangle,\ t_2 = \langle 10\rangle\}$.

As an example of a propagator consider the equality operation. When propagating from operands to result, the rule is: if both operands' bits are known and pairwise the same, the result is true. If any of the bits are different, the result is false. When propagating from the result to operands, there are two new rules: (1) If the result is fixed to 1, then any fixed bits of one operand should be the same as the corresponding bits of the other. (2) If the result is fixed to 0, and there is a single $\star$ value, and all the other bits are fixed to the same values, then that $\star$ value should be fixed to the negation of the value in the same position of the other operand; for example, given $(\langle 0 \star 0\rangle =^f \langle 000\rangle)$, the $\star$ value should be fixed to 1.

If a propagator discovers an inconsistent assignment, it will set the partial assignment to empty ($\mu = \perp$) which halts propagation. For instance, the partial assignment will be set to $\perp$ when processing the sub-expression: $\langle 0\rangle + \langle 1\rangle =^t \langle 0\rangle$, which means that the entire expression, not just that sub-expression, is unsatisfiable.

For convenience, in the rest of this chapter we use a shorter notation for the extract operation, using x_i to mean $x[i]$.

Later we will discuss in detail propagators for the addition and multiplication operations. However, we start by describing in detail the bit-vector and propagator.

initial	relations	result
$0 \wedge \star = \star$	$0 \wedge 1 = 0, 0 \wedge 0 = 0$	$0 \wedge \star = 0$
$1 \wedge 1 = \star$	$1 \wedge 1 = 1$	$1 \wedge 1 = 1$
$0 \wedge 1 = \star$	$0 \wedge 1 = 0$	$0 \wedge 1 = 0$
$0 \wedge 0 = \star$	$0 \wedge 0 = 0$	$0 \wedge 0 = 0$
$\star \wedge \star = 1$	$1 \wedge 1 = 1$	$1 \wedge 1 = 1$
$1 \wedge \star = 1$	$1 \wedge 1 = 1$	$1 \wedge 1 = 1$
$0 \wedge \star = 1$	$\{\}$	$\perp$
$0 \wedge 1 = 1$	$\{\}$	$\perp$
$0 \wedge 0 = 1$	$\{\}$	$\perp$
$1 \wedge \star = 0$	$1 \wedge 0 = 0$	$1 \wedge 0 = 0$
$1 \wedge 1 = 0$	$\{\}$	$\perp$

Table 4.1: Given the input on the left hand side, the result is the rightmost column. We give only the rules that cause a change. Only some of the 27 possible permutations are shown.

4.3 A “Bit-Vector And” Propagator

In this section we take the simple “bit-vector and” (bvand) operation and define its propagator formally. Later in the chapter we focus on more difficult operations which we present less formally.

A bvand of bit-width n takes the logical ‘and’ of two n-bit operands giving a result. It takes two operands ($v_0^{[n]}$, $v_1^{[n]}$) and gives one result ($v_2^{[n]}$). Let D be a mapping from each variables’ bits to a **3** value.

The bvand operation is a bit-wise operation, the value at position k of some bit-vector, depends only on the values at k of the other bit-vectors. So it’s enough to show the properties for just a single bit.

The operation is commutative, so we give only some permutations in Table 4.1. The ‘initial’ column contains the initial assignments to variables, the relations column gives the valid assignments that when joined produce the ‘result’.

The initial state summarises a set of relations. We show some of the relations that are summarised in the “relations” column. Taking the least upper join of the relations gives the result. For every $\star$ value in the result, there exists at least two relations in the set, where those relations have a 1 and 0 in the same position as the $\star$.

4.4 Some Useful Propagators

For convenience, we implement the right shift propagator by reversing the first operand and the result, then using left shift propagator, then reversing the first operand and the result again. This requires a simple optimal reverse propagator. This reverse propagator swaps each bit at position i with the bit at position $n - i$, where n is the bit-width. Because of the extra reversing steps involved, our right shift propagator runs slightly slower than our left shift propagator.

If operands to a propagator are the same, then extra information may be determined. Consider the expression $(t + t)$. A propagator that recognises that the two operands are the same can determine that the result must be even. Our propagators do not consider such aliasing. However, in most cases, instances that would benefit from aliasing have already been removed during normalisation, for example, replacing $(t + t)$ by $(2 \times t)$.

4.4.1 An Addition Propagator

Our addition propagator performs an interval analysis to estimate (the minimum and maximum of) the addends that may be 1, for each column. One of our multiplication propagators (presented in section 4.4.2) also uses this same approach. We use intervals because they allow simple reasoning about the full adder's majority and parity functions.

Let x_i, y_i, and c_i be the single bit addends of column i. The c variables are created by the addition propagator, and are not used outside it. We call c_i the carry-in to the column and c_{i+1} the carry-out. c_0 is set to $\langle 0 \rangle$ whereas c_n, where n is the bit-width, is ignored. The result bit is $r_i = x_i \oplus y_i \oplus c_i$. The carry-out is the majority function: $c_{i+1} = (x_i \wedge c_i) \vee (y_i \wedge c_i) \vee (x_i \wedge y_i)$.

The details of the operation are shown in Algorithm 4.1. For each column we calculate a lower bound l and an upper bound u of the number of elements of $\{x, y, c_{in}\}$ which may possibly be 1. The propagator works one column at a time, generally moving from less significant bits towards more significant bits. However, if some carry-in is updated in the process, then propagation moves back to the prior column (if it exists).

We do not have a formal proof that the addition propagator is maximally precise. However, we show in Table 4.5 that it is maximally precise from a bit-width of 1 to 5. The full-adder at bit l depends just on the carry-in from bit $l-1$, the carry-out to bit $l+1$, one bit of each operand, and the resulting bit. Because each full-adder is local and interacts with its immediate neighbours only, it should be possible to use this property to construct an inductive proof of precision. We leave this, however, for future work.

Example 4.1

Consider applying Algorithm 4.1 to a column i in the context $\mu = \{x_i \mapsto 1, y_i \mapsto \star, c_i \mapsto \star, r_i \mapsto 1, c_{i+1} \mapsto 1\}$.

Initially $l = 1$ and $u = 3$; next, because c_{i+1} is 1, l is set to 2; and next, because r_i is odd, l is incremented to 3. Now $l = u$, so in line 25, $\mu(y_i) \leftarrow 1$ and $\mu(c_i) \leftarrow 1$. Finally, because the c_i value has changed, propagation moves back to the prior column. ■

Example 4.2

For a more complex example, involving both left and right sweeps across the bit-vectors involved, consider $x = \langle 000001\star\star \rangle$, $y = \langle 00000\star 1\star \rangle$, and $r = \langle \star\star\star\star 111\star \rangle$. For $i = 0, 1, 2$, the body of Algorithm 4.1's while loop does nothing. For $i = 3$, however, the carry-in gets determined. More specifically, we find that $l = u = 1$, and consequently c_3 is determined to be 1. This causes the algorithm to revisit the previous column ($i = 2$), this time finding a larger lower bound $l = 3$. But this means all bits in that column are 1, that is, $y_2 = c_2 = 1$. Similarly, now that the carry-in c_2 has been determined, attention shifts to column 1, where it is determined that $x_1 = c_1 = 1$. For column 0, propagation is particularly effective: From the outset we had no knowledge of any of x_0, y_0, and r_0. The propagation method finds $l = u = 2$, which means we must have $x_0 = y_0 = 1$ and $r_0 = 0$.

After this backwards sweep, the algorithm again proceeds right-to-left, picking up at column 3, where c_4 is determined as being 0. For columns 4 to 7, the remaining unknown bits are then easily determined: clearly $r_4 = r_5 = r_6 = r_7 = 0$. At this point the algorithm stops, having determined all bits. ■

Algorithm 4.1 Propagating information about the addition relation for an addition $x^{[n]} + y^{[n]} = r^{[n]}$.

```
Require: $x^{[n]}, y^{[n]}, r^{[n]}$, lists of ternary variables
Require: $ones(i)$ yields number of $x_i, y_i, c_i$ that are 1
Require: $nonzeros(i)$ yields number of $x_i, y_i, c_i$ that are 1 or $\star$
 1: Create $c^{[n+1]}$, a list of ternary variables
 2: Initialise all $c$ variables to $\star$
 3: $c_0 \leftarrow 0$
 4: Create integers $i, l, u$
 5: $i \leftarrow 0$
 6: while $i < n$ do
 7:     $l \leftarrow ones(i)$
 8:     $u \leftarrow nonzeros(i)$
 9:     if $c_{i+1} = 1$ then $l \leftarrow max(2, l)$ end if
10:     if $c_{i+1} = 0$ then $u \leftarrow min(1, u)$ end if
11:     if $r_i = 1$ and $l$ is even then increment $l$ end if
12:     if $r_i = 1$ and $u$ is even then decrement $u$ end if
13:     if $r_i = 0$ and $l$ is odd then increment $l$ end if
14:     if $r_i = 0$ and $u$ is odd then decrement $u$ end if
15:     if $l \geq 2$ then $c_{i+1} \leftarrow 1$ end if
16:     if $u \leq 1$ then $c_{i+1} \leftarrow 0$ end if
17:     if $u < l$ then return $\perp$ end if
18:     $c' \leftarrow c_i$
19:     if $l = u$ then
20:         $r_i \leftarrow$ the parity of $l$
21:         if $ones(i) = l$ then
22:             set each addend that is $\star$ to 0
23:         end if
24:         if $nonzeros(i) = l$ then
25:             set each addend that is $\star$ to 1
26:         end if
27:     end if
28:     if $c' \neq c_i \land i > 0$ then
29:         decrement $i$
30:     else
31:         increment $i$
32:     end if
33: end while
```

An equivalent way to reason about the equations is using the truth tables for **3** (Figure 4.2). For instance, using the result bit equation, if there is only a single $\star$ value, re-arranging the equation to isolate the $\star$ value variable gives the value it should take.

The algorithm is linear in n: once some column i has been revisited owing to c_{i+1} having been set, it will not be revisited again (except in the course of the overall right-to-left sweep).

The soundness of Algorithm 4.1 follows from each step in the algorithm being correct. Termination is somewhat less clear, because i may grow or shrink in the body of the while loop. However, for successive iteration steps, consider the pairs $(s, n - i)$ of natural numbers, where s is the number of bits in x, y, r, and c that are undetermined ($\star$). It is easy to see that each iteration strictly decreases $(s, n - i)$ ordered lexicographically. That is, either some bits are determined (that is, s decreases), or else s remains unchanged, and as a result of the lack of change, i is incremented. Termination follows, since $\mathbb{N}^2$, ordered lexicographically, is well-founded.

4.4.2 Multiplication Propagators

In this section we describe two propagation methods that we combine to produce our multiplication propagation solver. The combination is an efficient, albeit not optimal, propagator. The two complement each other: for each method there are instances where it can fix bits that the other cannot.

The first propagator enforces consistency over the number of trailing zeroes of the operands. The second propagator is a more general version of the addition propagator already described. Instead of performing an exclusive-or over just three addends, we apply it over an arbitrary number of addends.

Consistency of the Number of Trailing Zeroes

This propagator exploits the fact that, given $x \times y = r$, the sum of trailing zeroes of x and y equals the number of trailing zeroes in r. If the sum is greater than, or equal to, the bit-width, then the result will be zero.

For proof consider that x can be written as $v \times 2^l$, and y can be written as $w \times 2^m$, where v and w are odd, and where l and m are the number of trailing zeroes in the respective operands. Zero can be written as (1×2^n). The result of multiplying the values is $v \times w \times 2^{l+m}$. Because both v and w are odd, the bit in position $m + l$ of the result will be 1, and all less significant bits in the result are 0.

Algorithm 4.2 Enforcing a consistent number of trailing zeroes on a multiplication's x operand where $x \times y = r$ and each variable is a vector of variables. For each position in x the algorithm checks if that position can be the rightmost 1 value, and if not, sets it to 0. We call it twice, once for x, y, r, and once for y, x, r.

```
Require: $x^{[n]}$, $y^{[n]}$, $r^{[n]}$, lists of ternary variables
 1: $ty$ = index of the least significant possible 1 value of $y$, (that is, $y_{ty} = 1$ or
    $y_{ty} = \star$). If $y = 0$, then $ty = n$.
 2: $tr$ = index of the least significant 1 value of $r$.
 3: $min = min(ty, tr, n)$.
 4: for $i \in 0..(n-1)$ do
 5:     if $x_i = 1$ then return
 6:     else if $x_i \neq 0$ then
 7:         for $j \in 0..min$ do
 8:             if $(j + i \geq n) \vee (y_j \neq 0 \wedge r_{i+j} \neq 0)$ then return
 9:             end if
10:         end for
11:         $x_i \leftarrow 0$
12:     end if
13: end for
```

Our propagator starts from the least significant bit of an operand and checks if the bit can be 1; if it can, we stop. Otherwise, it sets the variable to 0 and continues checking. The method is shown in Algorithm 4.2. We shall show an example of running this propagator shortly.

Setting bits in the result r is performed separately. The number of trailing zeroes in both x and y is summed, and that many trailing bits in r are set to zero.

Let us give some examples of reasoning about the trailing-zeroes, where we assume the constraints are asserted at the top level.

Example 4.3

Consider the initial constraint $(\langle 111\star\rangle \times \langle 11\star\star\rangle = \langle 1000\rangle)$. This gets strengthened to $(\langle 1110\rangle \times \langle 1100\rangle = \langle 1000\rangle)$. The last bit of the first operand cannot be 1, because then the result would have at most two trailing zeroes, so it is set to 0. Similar reasoning holds for the second operand. Since this is the first example, let us trace in detail how Algorithm 4.2 proceeds:

Consider $x = \langle 111\star\rangle$, $y = \langle 11\star\star\rangle$, $r = \langle 1000\rangle$. Initially $min = min(2, 3, 4) = 2$, and soon we have $i = 0$, and $j = 0$. The first time line 8 is reached, the test is $(0 + 0 \geq n) \vee (y_0 \neq 0 \wedge r_{0+0} \neq 0)$, which fails. The second time, when $j = 1$, the test is $(1 + 0 \geq n) \vee (y_1 \neq 0 \wedge r_{0+1} \neq 0)$, which also fails. Again, the third time, when j has reached min, $(2 + 0 \geq n) \vee (y_2 \neq 0 \wedge r_{0+2} = 0))$ fails. Hence x_0 is set to 0, and attention turns to x_1. Because $x_1 = 1$, we return (line 5). Note that x is now fully determined.

Next x and y are swapped, and the algorithm is called again. So let $x = \langle 11 \star \star \rangle$, $y = \langle 1110 \rangle$, $r = \langle 1000 \rangle$. Initially $min = 1$, and soon we have $i = 0$, $j = 0$. The first time line 8 is reached, the test is $(0 + 0 \geq n) \vee (y_0 \neq 0 \wedge r_{0+0} \neq 0)$, which fails (as $y_0 = 0$). The second time, when $j = 1$, the test is $(1 + 0 \geq n) \vee (y_1 \neq 0 \wedge r_{0+1} \neq 0)$, and again this fails (as $r_1 = 0$). Hence x_0 is set to 0, and attention turns to x_1. We have $x_1 = \star$, and again the test at line 8 fails repeatedly, first because $y_0 = 0$, and next because $r_2 = 0$. So x_1 is also set to 0, and so all bits have been determined. ■

Example 4.4

Consider $(\langle 1\star \rangle \times \langle \star\star \rangle = \langle 00 \rangle)$. At the start of Algorithm 4.2, ty is set to zero, and tr is set to two. The algorithm will strengthen this to $(\langle 1\star \rangle \times \langle \star 0 \rangle = \langle 00 \rangle)$. This makes sense: If the last bit of the second operand was 1, there would be at most one trailing 0 in the result, so the bit must be 0. ■

Example 4.5

As another example of Algorithm 4.2, consider $(\langle \star \star \star \rangle \times \langle 0 \star 0 \rangle = \langle 10\star \rangle)$. First, because the second operand has one trailing 0, the result must have at least one trailing 0, which yields $(\langle \star \star \star \rangle \times \langle 0 \star 0 \rangle = \langle 100 \rangle)$. Second, if the last bit of the first operand is 1, then it is not possible to get exactly two trailing zeroes in the result. So it must be 0, yielding $(\langle \star \star 0 \rangle \times \langle 0 \star 0 \rangle = \langle 100 \rangle)$. Note that the second bit of both operands must be 1. The algorithm described in the next section sets these bits. ■

Bounds Consistency over Partial Products

The second multiplication propagator we implement is a generalised version of the addition propagator that we described in subsection 4.4.1. The implementation is complicated because it adds an arbitrarily large number of partial products.

Our bounds consistency propagator operates on a table of partial products (see Figure 4.3). The table of partial products contains a column for each bit of the

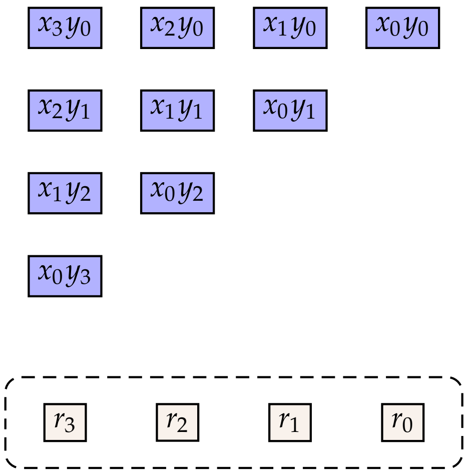


Figure 4.3: A 4-bit multiplication's table of partial products. Column zero is on the right.

output. In each column, one value is taken from each operand, where the sum of their indices equals the index of the column. The two values are conjoined. For example, column 2 contains $\{x_0 \wedge y_2, x_1 \wedge y_1, x_2 \wedge y_0\}$. The exclusive-or of these values, when combined with the carry-in, gives the resulting bit. However, the formula for the carry-in quickly gets complicated, so instead of summing using exclusive-or, *we use normal integer addition*, and take the parity of the result. Since information is partial, rather than working with integers, we deal with integer intervals. For this reason we refer to the technique as *column bounds propagation*.

For each column in the table of partial products, without considering that some partial products contain the same variables, the propagator establishes an integer interval (bounds) on both the *partial product count* (*ppc*) and the *sum* (which includes the carry).

Given three vectors of abstract variables x, y and r of bit-width n, where $x^{[n]} \times y^{[n]} = r^{[n]}$, we create a set of products (P_c) for each column number c, in $(0 \ldots (n-1))$. Let $P_c = \{(i, j) \mid i + j = c\}$. The *partial product count* is the number of partial products known to evaluate to 1, that is: $ppc_c = |\{(i, j) \in P_c \mid x_i = y_j = 1\}|$. We define $ppc_c^{\downarrow}$, to be the lower bound of the partial product count, that is, ppc_c evaluated with all $\star$ assignments ($\mu(v) = \star$), replaced by 0 ($\mu(v) \leftarrow 0$). The upper bound $ppc_c^{\uparrow}$ is the sum evaluated with all $\star$ values in the partial assignment replaced by 1. The *sum* is defined recursively: $sum_0 = ppc_0$, and $sum_c = ppc_c + \lfloor \frac{sum_{c-1}}{2} \rfloor$, for c in $\{1 \ldots (n-1)\}$. That is, the sum is the sum of the partial products in column c, together with all

Algorithm 4.3 Column bounds propagator for multiplication $x \times y = r$

```
Require: $x^{[n]}, y^{[n]}, r^{[n]}$, in the context of $x \times y = r$.
 1: $P_c = \{(i, j) \mid i + j = c\}$
 2: for $c \in \{0..n-1\}$ do
 3:     $ppc_c^{\downarrow} \leftarrow |\{(i, j) \in P_c \mid x_i = y_j = 1\}|$
 4:     $ppc_c^{\uparrow} \leftarrow |\{(i, j) \in P_c \mid x_i \neq 0 \wedge y_j \neq 0\}|$
 5: end for
 6: $sum_0^{\downarrow} \leftarrow ppc_0^{\downarrow}$
 7: $sum_0^{\uparrow} \leftarrow ppc_0^{\uparrow}$
 8: for $c \in \{1..n-1\}$ do
 9:     $sum_c^{\downarrow} \leftarrow ppc_c^{\downarrow} + \lfloor \frac{sum_{c-1}^{\downarrow}}{2} \rfloor$
10:     $sum_c^{\uparrow} \leftarrow ppc_c^{\uparrow} + \lfloor \frac{sum_{c-1}^{\uparrow}}{2} \rfloor$
11: end for
12: repeat
13:     for $c \in \{0..n-1\}$ do
14:         if $r_c \neq \star \wedge parity(sum_c^{\downarrow}) \neq r_c$ then
15:             increment $sum_c^{\downarrow}$
16:         end if
17:         if $r_c \neq \star \wedge parity(sum_c^{\uparrow}) \neq r_c$ then
18:             decrement $sum_c^{\uparrow}$
19:         end if
20:     end for
21:     Perform integer bounds propagation on $sum^{\downarrow}$, $sum^{\uparrow}$, $ppc^{\downarrow}$ and $ppc^{\uparrow}$ variables
        (Algorithm 4.4)
22:     if for some column $c$, $ppc_c^{\downarrow} > ppc_c^{\uparrow}$ or $sum_c^{\downarrow} > sum_c^{\uparrow}$ then
23:         set $\mu = \bot$ and return
24:     end if
25:     Do singleton interval propagation on $x$, $y$, and $r$ variables (Algorithm 4.5)
26: until all $x$, $y$ and $r$ bit values are stable
```

the carries that spill into that column. For this we likewise use lower and upper bounds $sum_c^{\downarrow}$ and $sum_c^{\uparrow}$.

When the lower and upper bounds of a sum coincide and sum_c is odd, then the result of that column is 1 (that is, $result_c = 1$). If it is even then $result_c = 0$.

We perform propagation on the intervals until they reach a fixed point. The detailed method is shown in Algorithm 4.3. Lines 1–5 initialise the *ppc* variables by counting the minimum and maximum number of partial products in each column. Lines 6–11 initialise the lower and upper bounds of the *sum*, for $i \in \{1..n\}$ (as usual the division rounds towards zero). Lines 12–26 is the workhorse of the algorithm which repeatedly tightens lower and upper bounds of the *sum* and *ppc* variables. First (lines 13–20), if the result bit of a column is known, then we enforce that the

Algorithm 4.4 Column bounds propagator: Integer bounds propagation

```
1: Apply propagation using the following propagators:
```

$$
\begin{array}{lcl}
sum_0^{\downarrow} & \leftarrow & max(sum_0^{\downarrow}, ppc_0^{\downarrow}) \\
sum_0^{\uparrow} & \leftarrow & min(sum_0^{\uparrow}, ppc_0^{\uparrow}) \\
\\
ppc_0^{\downarrow} & \leftarrow & max(sum_0^{\downarrow}, ppc_0^{\downarrow}) \\
ppc_0^{\uparrow} & \leftarrow & min(sum_0^{\uparrow}, ppc_0^{\uparrow}) \\
\\
sum_c^{\downarrow} & \leftarrow & max(ppc_c^{\downarrow} + \lfloor \frac{sum_{c-1}^{\downarrow}}{2} \rfloor, sum_c^{\downarrow}) \\
sum_c^{\uparrow} & \leftarrow & min(ppc_c^{\uparrow} + \lfloor \frac{sum_{c-1}^{\uparrow}}{2} \rfloor, sum_c^{\uparrow}) \\
\\
ppc_c^{\downarrow} & \leftarrow & max(sum_c^{\downarrow} - \lfloor \frac{sum_{c-1}^{\uparrow}}{2} \rfloor, ppc_c^{\downarrow}) \\
ppc_c^{\uparrow} & \leftarrow & min(sum_c^{\uparrow} - \lfloor \frac{sum_{c-1}^{\downarrow}}{2} \rfloor, ppc_c^{\uparrow}) \\
\\
sum_{c-1}^{\downarrow} & \leftarrow & max(2 \times (sum_c^{\downarrow} - ppc_c^{\uparrow}), sum_{c-1}^{\downarrow}) \\
sum_{c-1}^{\uparrow} & \leftarrow & min(2 \times (sum_c^{\uparrow} - ppc_c^{\downarrow}) + 1, sum_{c-1}^{\uparrow})
\end{array}
$$

lower and upper bounds of the *sum* have the same parity (the function *parity* is defined by $parity(k) = k \mod 2$). Second (line 21), bounds propagation is applied. We describe this shortly, and Algorithm 4.4 gives details. These last two steps are repeated until all lower and upper bounds for *sum* and *ppc* are stable.

Next (lines 22–24) possible inconsistency is detected, and finally (line 25) bit values are extracted in cases where lower and upper bounds of intervals coincide. The details of this are provided as Algorithm 4.5. There are three steps involved. First, if the lower and upper bound of a column's *sum* coincide then the result bit for that column is determined (Algorithm 4.5's lines 2–4). Second, if the lower and upper bounds of a column's *ppc* are the same and there are already enough ones in the column, then any partial product of form $\star \times 1$ in that column must in fact be 0×1, and similarly for a partial product of form $1 \times \star$ (lines 5–14). And third, dual to the last case, if the lower and upper bounds coincide and every partial product which *could* yield 1 in fact *must* yield 1, then we can change the $\star$ values to 1 (lines 15–24).

The steps of Algorithm 4.3 just described may set bits of x, y and/or r. Hence the whole process is repeated, until no new bit values are deduced. Again, the outermost repeat loop is guaranteed to terminate, as the only changes to bit values

Algorithm 4.5 Column bounds propagator: Fixing bits by singleton interval propagation

```
for $c \in \{0..n-1\}$ do
    if $sum_c^{\downarrow} = sum_c^{\uparrow}$ then
        $r_c \leftarrow parity(sum_c^{\downarrow})$
    end if
    if $ppc_c^{\downarrow} = ppc_c^{\uparrow} = |\{(i,j) \in P_c \mid x_i = 1 \wedge y_j = 1\}|$ then
        for $(i,j) \in P_c$ do
            if $\mu(x_i) = \star \wedge \mu(y_j) = 1$ then
                $\mu(x_i) \leftarrow 0$
            end if
            if $\mu(x_i) = 1 \wedge \mu(y_j) = \star$ then
                $\mu(y_j) \leftarrow 0$
            end if
        end for
    end if
    if $ppc_c^{\downarrow} = ppc_c^{\uparrow} = |\{(i,j) \in P_c \mid x_i \neq 0 \wedge y_j \neq 0\}|$ then
        for $(i,j) \in P_c$ do
            if $\mu(x_i) = \star \wedge \mu(y_j) \neq 0$ then
                $\mu(x_i) \leftarrow 1$
            end if
            if $\mu(x_i) \neq 0 \wedge \mu(y_j) = \star$ then
                $\mu(y_j) \leftarrow 1$
            end if
        end for
    end if
end for
```

involved replace $\star$ values by 0 or 1. The integer bounds propagation implemented by Algorithm 4.4 was inspired by similar bounds propagation methods in constraint programming [MS98] and CSP techniques. It propagates lower and upper integer bounds for column sums, both from less significant columns to more significant columns, and vice versa. Note that this part of the process must terminate because the steps involved can only tighten, never relax, bounds. The following example shows how the propagation works.

Example 4.6

Consider the situation where $sum_4 = [2,5]$, $ppc_5 = [1,1]$, $sum_5 = [1,2]$, and $r_5 = \star$. Here $[a,b]$ means that a is the current lower bound, and b the upper bound. As the result bit is unknown, lines 12–20 of Algorithm 4.3 will not change the sum.

Next the propagators of Algorithm 4.4 are applied to the equation $[1,2] = [1,1] + \lfloor\frac{[2,5]}{2}\rfloor = [1,1] + [1,2] = [2,3]$. Note that $sum_i^{\downarrow} = max(ppc_i^{\downarrow} + \lfloor\frac{sum_{i-1}^{\downarrow}}{2}\rfloor, sum_i^{\downarrow})$ when instantiated reads $sum_5^{\downarrow} = max(1 + 2/2, 1)$, so the lower bound of sum_5 tightens from 1 to 2. Also note that $sum_{i-1}^{\uparrow} = min(2 \times (sum_i^{\uparrow} - ppc_i^{\downarrow}) + 1, sum_{i-1}^{\uparrow})$ when instantiated reads $sum_4^{\uparrow} = min(2 \times (2 - 1) + 1, 5)$, so the upper bound of sum_4 tightens from 5 to 3.

Substituted into the definition of sum_5, the final bounds are $[2, 2] = [1, 1] + \lfloor \frac{[2,3]}{2} \rfloor = [1, 1] + [1, 1] = [2, 2]$. Since the lower and upper bound of the sum are both 2, Algorithm 4.5 (line 3) will set $r_5 = 0$. ■

Our column bounds propagator is powerful; it subsumes each of the following three natural multiplication propagators.

First, it subsumes the propagator that sets some of the most significant bits of the result to zero when the multiplication of x and y cannot cause overflow. That is, in all positions i where $2^i > x^{\uparrow} \times y^{\uparrow}$, set r_i to 0. It is not hard to see that our column bounds propagator subsumes this. When x and y take their maximum values, the sum of each column equals $sum_i^{\uparrow}$. Eventually the upper bound of the sum will go to zero, and when it does, the upper and lower bound of the sum will both be zero, hence the result bit will be set to zero.

Example 4.7

Applying column bounds propagation (Algorithm 4.3) to $(\langle 000 \star \star \rangle \times \langle 000 \star \star \rangle) = r^{[5]}$, at line 25 we have $sum_0 = [0, 1], sum_1 = [0, 2], sum_2 = [0, 2], sum_3 = [0, 1], sum_4 = [0, 0]$. So r_4 is set to 0. Note that, when the operands take their maximum possible values, so we have $(00011)_2 \times (00011)_2$, the number of true partial products in each column including carries, equals the sums' upper bounds. ■

Second, our propagator subsumes the propagation principle for multiplication that, if the least significant j bits of both the operands are known, then the least significant j bits of the result are uniquely determined. For our column bounds propagator, at column j, when all the bits of the operands in positions less than or equal to j are fixed, there will be no $\star$ values in partial products of those columns. Without $\star$ values, the ppc in each of those columns is exact, so the sum in each of those columns is known, and by taking the parity of the sum, the result bit is calculated.

Example 4.8

Applying column bounds propagation to $(\langle\star 01\rangle \times \langle\star 10\rangle = r^{[3]}$, at line 25 we have $sum_0 = [0,0], sum_1 = [1,1], sum_2 = [0,1]$. So r_0 is set to 0, and r_1 is set to 1. ■

Third, our propagator subsumes the propagator that, when some of the least significant bits of the result and one operand are known, the multiplicative inverse (when it exists) of the partially known operand can derive extra known bits of the other operand.

Example 4.9

Consider $\langle\star\ \star\ 11\rangle \times \langle\star\ \star\ \star\star\rangle = \langle\star 110\rangle$. We can restrict our attention to the right-most sections where one of the operands and the result are entirely fixed: $\langle 11\rangle \times \langle\star\star\rangle = \langle 10\rangle$. The multiplicative inverse of 3 modulo 2^2 is 3, that is, (3×3) mod $4 = 1$. Multiplying the result by this inverse gives 2, so we can update the second operand to $\langle\star\ \star\ 10\rangle$. ■

It is well known that x has a multiplicative inverse modulo 2^n if and only if x is odd.

Again, column bounds propagation subsumes this. If the least significant w bits of both $r^{[n]}$ and $x^{[n]}$ are fixed, and x is odd, then bits $(w-1)\ldots 0$ of $y^{[n]}$ will be fixed, as can be seen by a simple proof by cumulative induction.

- For the base case, in column 0 we have a single addend, x_0y_0, and by assumption $x_0 = 1$ (x is odd). Hence the *sum* will be [0,1]. Because r_0 is known, the interval will be tightened. If r_0 is 1, it is tightened to $[1,1]$, and y_0 will be set by Algorithm 4.5 at line 21. Otherwise, first the sum and then *ppc* will be tightened to $[0,0]$, and y_0 will be fixed by Algorithm 4.5 at line 11. Hence y_0 is determined, in fact equal to r_0.

- Now assume that y_j is determined for all $j < k$. We show that y_k must be determined. The addends of column k are $x_ky_0, x_{k-1}y_1, \ldots, x_1y_{k-1}, x_0y_k$. But since $y_0, \ldots y_{k-1}$ as well as $x_0 = 1$ are determined, each of these addends is 0 or 1, except the last, which is y_k. That is, the upper and lower bound of the *sum* differ by at most 1. Because the result bit is known, the bounds will be tightened by Algorithm 4.5. That is, the equation for sum_k boils down to $c + y_k = r_k$ for some integer constant c. Hence the propagation algorithm will determine y_k, setting y_k to r_k's value (if c is even) or to its complement (if c is odd).

From this it follows that the column bounds propagator will fix at least as many bits as the rule that exploits multiplicative inverses.

Example 4.10

Consider $x \times y = r$, with $x = \langle\star\star 1\rangle$, $y = \langle 011\rangle$ and $r = \langle 001\rangle$. When we process the 0th column, we begin with $x_0 \wedge y_0 = r_0$, which after substituting in known values gives $1 = 1$.

When we process the 1st column, we begin with $x_0 y_1 \oplus x_1 y_0 = r_1$, which after substituting in known values gives $y_1 \oplus 1 = 0$, which sets y_1 to 1.

When we process the 2nd column, we begin with $y_2 x_0 \oplus y_1 x_1 \oplus y_0 x_0 = 0$, which after substituting gives $y_2 \oplus 1 \oplus 1 = 0$, which sets y_2 to 0. So $y^{[3]}$ is set to $\langle 011\rangle$, which indeed gives the multiplicative inverse of 3, that is, $(3 \times 3) \equiv_8 1$. ■

Example 4.11

A simple example for which the propagator is not optimal, is: $x^{[2]} \times y^{[2]} = r^{[2]}$ with $\mu = \{x = \langle 1\star\rangle, y = \langle 1\star\rangle, r = \langle 1\star\rangle\}$. Substituted into the Boolean formula definition of multiplication gives $r_0 = (x_0 \wedge y_0)$, $1 = x_0 \oplus y_0$. Since neither x_0 nor y_0 can be 1, r_0 must be 0. Our propagator does not deduce that the result must be even, because it conservatively treats the variables in the partial products of each column as being distinct. ■

The column bounds propagator generally subsumes interval propagation, but not quite. The reason it may fail is that multiplication is signedness-agnostic. For example, $\langle 111\star\rangle \times \langle 111\star\rangle = \langle\star\star\star\star\rangle$, interpreted as signed intervals is: $([-2,-1] \times [-2,-1]) = [-8,7]$, which can be strengthened to $([-2,-1] \times [-2,-1]) = [1,4]$. Hence the most significant bit of the result must be 0. However, the bounds analysis does not determine this.

4.4.3 An Unsigned Division Propagator

We propagate unsigned division by using a truncating integer division propagator that operates on unsigned integer bounds.

We begin by converting the operands and results from **3** to the integer bounds domain by calculating for each value the maximum and minimum value that the ternary variable contains. Next we enforce bounds consistency over those integer domains. Then we convert back to **3**. We perform these three steps until a fixed point is reached and the representation in **3** is stable. This terminates because we do not allow known values to become $\star$.

Like the multiplication propagator, this propagator is not optimal. As a simple example of non-optimality, the propagator is not able to deduce that the numerator must be odd in this case: $(\langle\star\star\rangle \div_u \langle\star 1\rangle) = \langle\star 1\rangle$. Interpreted as unsigned intervals this says $([0,3] \div_u [1,3]) = [1,3]$. Now it is not possible for the numerator to be 0, so the interval can be tightened to [1,3]. It is possible for each interval to take its extreme value, so no further propagation is possible. Converting the intervals back to the **3** domain leaves the ternary variables unchanged. However, if the numerator takes the value of 2, the denominator can be 1 or 3. So, the result must be either 0 or 2, neither of which the result can express. So, it is impossible for the numerator to be 2, so it can be either 1 or 3, so the least significant bit of the numerator must be 1.

Note that it is not straightforward to utilise the multiplication propagator for integer division. To turn $(a \div_u b) = q$ into $a = bq + r \wedge (b \neq 0 \Rightarrow r < b)$ is unattractive, and it is more profitable to utilise the fact that unsigned division cannot overflow, whereas multiplication can. The Beaver bit-vector solver (subsection 3.23.6) performs this same transformation in another context. As we show in the section 4.7, because unsigned division does not overflow, our division propagator is quite effective and able to determine many of the available bits.

Analysing multiplication using this interval approach is not practical because of the prevalence of overflow.

4.5 A Propagation Solver

The previous section described a number of propagators for various bit-vector operations. A propagation solver runs propagators until a global fixed point is reached. When each propagator is at a fixed point, the global fixed point has been reached. Initially the partial assignment of all the nodes' bits is $\star$, then the partial assignment of constants is updated to be the respective value. A propagator is only run when the partial assignment to its operands or output change.

In an attempt to reduce the number of times that expensive propagators are run, we use a fast and a slow worklist. Propagators that are fast, such as the bit-vector exclusive-or propagator, are run before expensive propagators such as the signed remainder propagator. We show later, for instance in Table 4.3d, that the exclusive-or propagator can be hundreds of times faster than the signed remainder propagator.

We run the propagation engine twice to a global fixed point. The first time we run propagation using only bit-vector constants or 1/0 as sources of fixed bits. In this phase, information never flows downwards from results to operands. Initially, all the propagators that depend on the constant values are added to the worklist. Propagators are taken from the work list one by one and run. If a propagator changes any ternary assignment, then all the propagators that depend on that assignment are added into the worklist. The propagation engine continues until the worklist is empty. When the worklist is empty, a global fixed point has been reached.

The second time that the propagation engine is run, the root node is set to true, and again we propagate until a global fixed point.

Once propagation reaches a global fixed point, how the results of the analysis can be used depends on what was assumed before propagation started. We discuss using the results of the analysis in the next section.

4.6 Using the Results

After the fixed point is reached, simplifications to the expression are applied—hopefully saving time overall. After bit propagation we use the partial assignment in three different ways. First, some expressions are replaced by the values discovered. Second, some values are replaced and an equality conjoined at the top level. Third, individual bits are conjoined with the CNF.

In the first case, before the root node is set to true, if an expression is found to have a particular value, then the node is replaced by that value. For instance, given the expression $((0^{[3]} :: x^{[2]}) = 7^{[5]})$, bit propagation will discover that the equality expression is necessarily 0, so the formula can be replaced by 0. With the root node *not* set to 1, bit propagation gives the values that nodes always take. There can never be a conflict, and variables' bits will never be fixed (because information only flows upwards).

When bit propagation is performed with the root node set to 1, expressions are replaced by constants and the expression conjoined to the top. Consider the sub-expression: "$(t_0 =^t t_1) \wedge^t p$". It is unsound to replace this with 1, because the condition that t_0 must equal t_1 is lost. Instead, this sub-expression is replaced by 1, and $(t_0 = t_1) \wedge p$ is conjoined with the root node.

Nodes that are partially set are stored and conjoined with the CNF expression of the formula just before sending it to the SAT solver. As an example, say the analysis reveals that $t = \langle 1 \star 1 \star 0 \rangle$. After bit-blasting, there is a Boolean formula produced for each element of this bit-vector that produces the value of the node. When the CNF encoding is run, each of these Boolean formulae is made equivalent to some fresh variable, say $t = \langle b_4, b_3, b_2, b_1, b_0 \rangle$. For each value that we know must be set to either 1 or 0, we assert the appropriate literal. So in our example, we add three clauses to the CNF: $\{\neg b_0, b_2, b_4\}$. These clauses fix the value of the SAT solver's variables, simplifying other clauses that contain the same variables.

4.7 Evaluation of Theory-Level Bit Propagation

We compare STP2 r1611 with and without theory-level bit propagation, and for reference compare against the current version of the SMT-COMP 2011 (QF_BV) winner Z3 3.2 [dMB08b] with and without bit propagation.

To isolate the effect of bit propagation, we created a standalone executable that reads SMT-LIB2 format, applies bit propagation, then outputs the simplified result. We used this to pre-process input to Z3. The processor applies two of the three techniques for simplifying expressions, it does not conjoin information about partially specified sub-expressions. In STP2, partial information about sub-expressions is used to add extra information.

We perform the evaluation with the same experimental configuration as in section 3.19. We took the SMT-LIB QF_BV benchmark set as of January 2012. We discarded the *asp* family benchmarks which is large (29GB), and contains encodings of problems we are uninterested in, for example: towers of Hanoi, travelling salesperson, and Sudoku problems. We discarded the *mcm* family because it uses syntax that STP2 cannot yet parse. We discarded the *bruttomesso:core* family as it contains no arithmetic, a key part of the software verification benchmarks we are interested in. We limited each family to 50 randomly chosen benchmarks and only chose a benchmark if some solver required more than 1 second to solve it. We were left with 715 benchmarks in 31 families. Finally, we used a memory limit of 3GB and a time limit of 500 seconds on a single core of an Intel E5507 Linux computer to run the benchmarks.

When multiple solvers returned a result for a benchmark, they always agreed about the result. Moreover, all the results agreed with the expected status as given by annotations in the benchmarks.

The results are shown in Table 4.2. For each family and solver, the number of failures and the number of those failures due to exceeding the memory limit is given. For each benchmark family, the best result (fewest failures) is highlighted with boldface type.

Z3 3.2 with bit propagation has the fewest failures of the solvers we compare, 10 fewer than with bit propagation disabled. Of the solvers we compare, STP2 with bit propagation is the best on the most families: 18.

Compared to no bit propagation, bit propagation enables 10 extra benchmarks to be solved. This is true for both STP2 and Z3, although the gains for the two are on different problems.

For STP2 to perform bit propagation on the 715 problems takes 65 seconds. When preprocessing Z3's input, bit propagation takes 210 seconds. The difference is because as a pre-processor many simplifications have not been applied, so bit propagation operates on a larger expression. Z3 3.2 with bit propagation has the lowest overall time and the fewest failures.

4.8 Testing that Propagators Are Optimal

In this section we describe how we generated the evidence that (most of) our propagators are optimal. We use the details in this section to later address the

		STP2 r1611		STP2+bp		Z3 3.2		Z3+bp	
Family	#	time	fail	time	fail	time	fail	time	fail
VS3	11	0	1/11	0	1/11	**548**	**7**	847	7
brummayerbiere	28	**231**	**1/12**	271	1/12	583	2/13	707	2/13
brummayerbiere2	50	2593	17	**1812**	**14**	2626	1/28	1388	1/18
brummayerbiere3	50	1078	31	**1832**	**29**	1698	30	1476	30
bruttomesso:lfsr	50	6735	4	7353	5	862		**838**	
bruttomesso:simple_proc	50	2866	6	3508	5	1742	3	**2993**	**2**
calypto	17	10	12	9	12	**947**	**11**	31	13
galois	3	**0**	**3**	**0**	**3**	**0**	**3**	**0**	**3**
gulwani-pldi08	3	26		25		21		**9**	
pipe	1	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
rubik	6	872	1	838	1	89	1	**301**	
sage:app1	50	269		281		220		**187**	
sage:app12	14	20		**0**		**0**		**0**	
sage:app2	1	**0**		**0**		11		**0**	
sage:app7	6	**0**		**0**		9		10	
sage:app8	50	461		**20**		66		56	
sage:app9	50	541		**19**		60		47	
spear:cvs_v1.11.22	28	65		**59**		130		139	
spear:inn_v2.4.3	50	55		**46**		297		277	
spear:openldap_v2.3.35	5	7	3	**513**		0	5	0	5
spear:samba_v3.0.24	50	**128**		133		589		528	
spear:wget_v1.10.2	41	91		**85**		488		334	
spear:xinetd_v2.3.14	1	**0**		**0**		2		1	
spear:zebra_v0.95a	5	**3**		**3**		14		15	
stp	1	20		23		**11**		23	
stp_samples	22	4	2	3	2	**1**	**2**	4	2
tacas07	3	89	1	**331**		709		964	
uclid_contrib_smtcomp09	7	703	1	**929**		1893		1659	
uclid:catchconv	50	65		**49**		142		166	
uum	7	45	6	30	6	**11**	**6**	**11**	**6**
wienand-cav2008:Booth	5	82	4	82	4	**35**	**4**	42	4
Sum	715	17071	115	18267	105	13820	114	13065	104
Time incl. penalty		74686s		70872s		70934s		65169s	

Table 4.2: STP2 and Z3 performance with and without bit propagation. “STP2” is STP2 with bit propagation disabled. The number of benchmarks that failed is given for each family. The number of times the memory limit was reached is indicated, for example, 1/11 means 11 failures, one of which was a memory out. All times are in seconds. The times given are the sum of the times for the successful instances only. Limits of 500 seconds and 3GB are used. The bottom row gives the times with a penalty of 501 seconds counted for each failed problem. Times are measured on a single core of an Intel E5507 Linux computer.

question: Do the results from applying bit propagation improve if more precise propagators are used?

We test that our implemented propagators are sound by comparing their effects against the optimal propagator's effects. The propagators we implement should produce a superset of the effect of the optimal propagator. That is, if the optimal propagator fixes a bit, then all propagators should produce that same value for the bit, or $\star$. For propagators we expect to be optimal, the sets from both should be the same.

We used two techniques to generate (the effect of) the optimal propagator. First, at small bit-widths, we generate the effect by exhaustively generating operands, then applying an operation to those operands, and then storing the operands and the result in a set. To determine the effect of the optimal propagator for abstract variables, an algorithm searches through the stored tuples and finds any calculations that are contained in that set. It applies the abstraction function to each matching tuple, then applies the join operation to those, giving the result of the optimal propagator. If no matching concrete values are found, it returns $\perp$.

More formally, to calculate the effect of the optimal propagator on the function $f(x^{[n]}, y^{[n]}) = r^{[n]}$, where $x^{[n]}, y^{[n]}, r^{[n]}$ are lists of ternary variables:

- Apply the function f to all possible concrete operands, and store the tuple. For i and $j \in (0 \ldots 2^n - 1)$, add $(i, j, f(i, j))$ into a set S.
- Search through S for concrete values that match elements in the set. Collect elements $\langle s_0, s_1, s_2 \rangle$, where $(s_0 \in \mu(x) \wedge s_1 \in \mu(y) \wedge s_2 \in \mu(r))$.
- Apply the abstraction function α to each matching element.

For example, $(\langle 10 \star 0 \rangle \ll \langle \star \star \star\star \rangle) = \langle 1 \star \star\star \rangle$ matches the following tuples: $\langle (1000)_2, (0000)_2, (1000)_2 \rangle$, $\langle (1010), (0000)_2, (1010)_2 \rangle$, $\langle (1010), (0010)_2, (1000)_2 \rangle$. Applying the abstraction function gives: $(\langle 10 \star 0 \rangle \ll \langle 00 \star 0 \rangle) = \langle 10 \star 0 \rangle$, which is the result of the optimal propagator.

For each two-input propagator we exhaustively generated all combinations for 1 bit through to 6 bits. At 6 bits we checked all 3^{18} distinct combinations. For propagators we believed were optimal, we checked: that the propagator is idempotent, the propagator and optimal propagator return $\perp$ at exactly the same time, and that the resulting partial assignments are the same. This identified many, but not all of the defects we found in our implementations of the propagators.

$$\begin{aligned}
\mathit{approx}(\varphi) &= \mathit{app}(\varphi, \bot) \\
\mathit{app}(\varphi, r) &= \textbf{if } \mathit{unsat}(\varphi) \textbf{ then } r \\
& \textbf{else let } s = r \sqcup \alpha\{\mathit{model}(\varphi)\} \\
&\phantom{= \textbf{else}} \quad \textbf{in } \mathit{app}(\varphi \wedge \neg\gamma(s), s)
\end{aligned}$$

Figure 4.4: Finding the optimal propagator: The approach of Reps, Sagiv and Yorsh. [RSY04]

Both techniques we use to generate the effect of the optimal propagator generalise from concrete values to abstract values by moving up the lattice.

At larger bit-widths, exhaustively generating values is impractical, so instead we compared our propagators against the optimal propagator, produced using the approach proposed by Reps, Sagiv and Yorsh [RSY04] (Figure 4.4). They show how to produce the result of the optimal propagator for domains that satisfy the ascending chain condition, which **3** does. Their algorithm uses a decision procedure which produces a model. We refer to their algorithm as RSY.

The *approx* method of Figure 4.4 is called with φ where φ describes the tuples that satisfy a relation. If the set of tuples is empty, then $\bot$ is returned, otherwise a search occurs to find tuples that are not contained in r. When no such r exists, then r is the best result possible.

Intuitively, rather than taking the union of all the tuples in the set described by the abstract variables, like the exhaustive approach does, RSY searches for new tuples that cause the abstract variables to change. If there are k $\star$ values initially, then the algorithm will perform the abstraction function at most $k + 1$ times.

RSY first searches for a model, then for models which cause each of the bits to take the opposite value, that is, if they have been 1 in all prior models, to take a 0.

Example 4.12

Consider applying RSY to $\langle\star\rangle \times \langle\star\rangle = \langle 0\rangle$.

- Before calling the *approx* function, $\star$ values are replaced by fresh variables, setting φ to $\langle v_0\rangle \times \langle v_1\rangle = \langle 0\rangle$.

- This is encoded as CNF, and its satisfiability is checked. In this case it is satisfiable, so the partial assignment is set to, say, $s(v_0) \leftarrow 0, s(v_1) \leftarrow 1$.

- Next the algorithm searches for a model where v_0 or v_1 take a different value, that is for a solution to $(v_0 \times v_1 = 0) \wedge (v_0 \neq 0 \vee v_1 \neq 1)$.
- This is satisfiable. The returned model might be $v_0 = 0, v_1 = 0$. Now the abstraction function is applied to the model, and then the meet taken with the current partial assignment. That is: $s(v_0) = s(v_0) \sqcup \alpha(0), s(v_1) = s(v_1) \sqcup \alpha(0)$, giving $s = \{v_0 = 0, v_1 = \star\}$.
- Next the algorithm asks for a model where v_0 does not equal 0, that is: $(v_0 \times v_1 = 0) \wedge (v_0 \neq 0)$, which returns satisfiable, updating the partial assignment to: $s = \{v_0 = \star, v_1 = \star\}$.
- All of the variables are $\star$, so the next call to the SAT solver returns unsatisfiable, and the algorithm returns $v_0 = \star, v_1 = \star$

It is concluded that even the optimal propagator will not determine any bits for the example. ■

To further test propagators, we generated random tuples at bit-widths between 7 and 256 and tested that the result of our propagators is the same or a superset of the result from RSY.

Using this approach, an instance of a defect that we encountered at higher bit-width was that our implementation of left and right-shift relied on the 64-bit machine's semantics (using just the bottom 8 bits of the second argument) as distinct from the SMT-LIB semantics (use all the bits). So given a 64-bit value left shifted by a large number with many trailing zeroes, our defective implementation returned the same input, rather than zero as the `QF_BV` semantics dictates.

4.9 An Optimal 6-Bit Multiplication Propagator

The multiplication propagators that we have discussed so far are not optimal. In this section we describe a multiplication propagator that is optimal for the least significant n bits (we use $n = 6$). The idea we present is generally applicable. However, in practice it will often be too slow to be useful.

We start by exhaustively generating assignments. Then apply the multiplication propagators described in subsection 4.4.2, then compare the results with the optimal propagator. Clauses are constructed to assign values that were missed by the multiplication propagators, and added to a SAT solver.

Example 4.13

Given $(\langle 1\star\rangle \times \langle 1\star\rangle = \langle 1\star\rangle)$, the propagators we have introduced will fail to determine that the least significant bit of the result must be 0. Hence, we remedy this by generating the clause $(x_1 \wedge y_1 \wedge r_1) \rightarrow r_0$. Note that the left-hand side expresses the literals fixed prior to calling the multiplication propagator. ■

The algorithm we use is shown in Algorithm 4.6. Because implementations of two watched literals [MMZ+01] and unit propagation are so efficient, we precalculate clauses that give the effect of the optimal propagator when combined with the other propagators. We use the SAT solver, just with unit propagation, and with search as a multiplication propagator.

The algorithm compares the effect of the optimal propagator, to another propagator. Whenever the effects differ, clauses are generated that explain the difference. Immediately after generating the clauses that explain the difference, they are conjoined with the clauses that have already been discovered. The algorithm traverses from low to high bit-widths. At bit-width i where $i \neq 1$, the clauses for an optimal multiplication propagator at bit-width $i-1$ have already been generated.

This approach is ideal for multiplication because the same clauses can be used to propagate on the least significant bits irrespective of the bit-width. The clauses that we calculate for the least significant 6-bits can be applied to multiplications of lesser and greater bit-width.

Running Algorithm 4.6, for $n = 6$, produces 56,943 clauses. However, we did not remove all the redundant clauses, so a smaller number of clauses could have the same effect.

The clauses vary in length from 4 to 13 literals, so the probability of any given clause setting values in a random assignment to a multiplication, or detecting a conflict varies from about 10%, to .0005%. Each clause requires work to be performed at runtime, but might avoid time spent performing conflict analysis. How useful a clause is, depends on how often it is applied, and how much work it saves minus the cost of applying unit propagation to it.

Algorithm 4.6 Generating clauses for n-bit multiplication

```
Require: $n$ the bitwidth
 1: Create $\varphi$ , a set of CNF clauses
 2: $\varphi \leftarrow \emptyset$
 3: for $i \in 1 \ldots n$ do
 4:     Create $x^{[i]}$, $y^{[i]}$, $r^{[i]}$, lists of ternary variables
 5:     for all distinct assignments to: $\langle x^{[i]}, y^{[i]}, r^{[i]} \rangle$ do
 6:         repeat
 7:             perform trailing zero propagation
 8:             perform partial product bounds consistency
 9:             perform unit propagation of $\langle x, y, z \rangle$ on $\varphi$
10:             perform unit propagation of $\langle y, x, z \rangle$ on $\varphi$
11:         until at a fixed point
12:         $(\langle x_p, y_p, r_p \rangle \leftarrow maxPrecise(x, y, z))$
13:         if $\langle x_p, y_p, r_p \rangle \neq \langle x, y, z \rangle$ then
14:             Set *difference* to the bits fixed in $\langle x_p, y_p, r_p \rangle$, but not in $\langle x, y, z \rangle$
15:             for all $d \in$ *difference* do
16:                 Add to $\varphi$ the implication that the fixed bits of, $\langle x, y, z \rangle$ implies $d$
17:             end for
18:         end if
19:     end for
20: end for
21: Output $\varphi$
```

4.10 Propagator Evaluation

To measure whether optimal propagators improve upon the results we have already presented (Table 4.2), we applied RSY after applying our imprecise propagators (multiplication, division, and remainder). None of the benchmarks contain signed modulus. We placed a limit of 500 seconds on each call to RSY; 594 instances finished. Of the problems that finished, extra bits were only fixed in one case; however, these extra assignments did not reduce the time taken.

We compare the precision of our propagators versus the precision of STP2's CNF encoding with unit propagation in Table 4.3a–Table 4.3d. Both begin with exactly the same initial assignment. We compare the effect of propagation for varying levels of information content. We compare on 100,000 64-bit values, with a fraction of bits in the operands and result set to 0 or 1. For each run, we generated two random 64 bit operands, we generated assignments that are random and uniform over 0 and 1. Then we applied the operation to those, saving the result. At the 1% level, next we set 99% of the bits of the operands and the result to $\star$. Likewise at the other levels. For example, with 5% information content, 2.5% of bits can be expected to be set to

Operation	1%				
	UP time	prop. time	UP bits	prop. bits	%
signed greater than equals	0.06s	0.01s	4	8	50%
unsigned less than	0.04s	0.06s	6	9	66%
equals	0.09s	0.01s	333	333	100%
bit-vector xor	0.06s	0.08s	1880	1880	100%
bit-vector or	0.04s	0.07s	95567	95567	100%
bit-vector and	0.06s	0.08s	95245	95245	100%
right shift	0.13s	2.32s	1604489	1604711	99%
left shift	0.15s	2.27s	1615256	1615505	99%
arithmetic shift	0.13s	3.97s	762885	767828	99%
addition	0.04s	0.09s	41	42	97%
multiplication	0.36s	0.21s	1473	1473	100%
multiplication (max $n = 6$)	0.71s	4.34s	1437	1437	100%
unsigned division	3.71s	0.99s	898513	899446	99%
unsigned remainder	4.01s	9.94s	22	856	2%
signed division	0.17s	3.38s	642	26850	2%
signed remainder	0.26s	30.68s	8	286	2%

Table 4.3a: Comparison of unit propagation and bit-blasting at 1%. 100000 iterations at 64 bits.

0, 2.5% to 1, and 95% to $\star$. The time we give excludes the time to create the random assignments. 'UP time' is the time to run unit propagation on a pre-generated CNF, 'prop. time' is the time to run the propagators, 'UP bits' is the number of extra bits fixed after calling unit propagation. 'prop. bits' is the number of extra bits fixed by the propagators. '%' is the percentage of the bits fixed by the propagators that were fixed by unit propagation over the bit-blasted representation. Times are given in seconds and were measured on a single core of an Intel Q8400 Linux computer.

In Table 4.3a–Table 4.3d, the "multiplication (max $n = 6$)" entry corresponds to the propagator described in section 4.9.

For some operations, the CNF encoding assigned all possible values. Namely for equals, bit-vector exclusive-or, bit-vector or, and bit-vector and. That is, no initial assignments were found for which the CNF encoding of these operations was not optimal under unit propagation.

The results show that in general the CNF encoding with unit propagation propagates well. It determines > 80% of the assignments compared to our propagators.

Using unit propagation to obtain a 6-bit optimal propagator for multiplication ran 180 times slower than without (Table 4.3c), but discovered 30% more assignments. However, at 95% there was a large cost, taking 30 times longer than without

Operation	5%				
	UP time	prop. time	UP bits	prop. bits	%
signed greater than equals	0.10s	0.07s	202	260	77%
unsigned less than	0.14s	0.06s	172	243	70%
equals	0.06s	0.03s	7310	7310	100%
bit-vector xor	0.11s	0.11s	45700	45700	100%
bit-vector or	0.14s	0.14s	464102	464102	100%
bit-vector and	0.20s	0.11s	463171	463171	100%
right shift	0.41s	0.98s	4713026	4715109	99%
left shift	0.47s	0.78s	4716925	4718901	99%
arithmetic shift	0.49s	1.68s	4566641	4591888	99%
addition	0.16s	0.12s	875	1029	85%
multiplication	1.49s	0.33s	7805	7816	99%
multiplication (max $n = 6$)	3.90s	20.54s	7794	7815	99%
unsigned division	15.70s	3.63s	3034619	3038246	99%
unsigned remainder	15.16s	28.05s	1071	9213	11%
signed division	1.27s	18.55s	34820	1264498	2%
signed remainder	1.71s	121.96s	276	7407	3%

Table 4.3b: Comparison of unit propagation and bit-blasting at 5%. 100000 iterations at 64 bits.

for little gain. More work needs to be done to understand the trade-offs. Note that the "UP time" for both multiplication variants differs, even though the work is the same, because the "multiplication with UP" takes more of the processor's data cache.

As the percentage of bits that are assigned increases, unit propagation is more time consuming. At the 1% level, unit propagation for some operations is comparable in speed to the propagators, however, at 95% the propagators are substantially faster.

The random assignments we produced, for instance in Table 4.3c, are atypical for the shift operations. The second operand of the random assignments at 64-bits has a high probability of being greater than 64, making the result 0. Shifting random assignments, over-estimates how well the shift propagators will work on real problems.

The shift operations run faster as the number of bits fixed increases (Table 4.3a–Table 4.3d), because the more bits that are assigned, the greater the probability that a bit is fixed that sets the result to zero. Setting the result to zero is fast to detect and perform.

Operation	50%				
	UP time	prop. time	UP bits	prop. bits	%
signed greater than equals	1.05s	0.30s	17551	22026	79%
unsigned less than	1.05s	0.23s	17577	22045	79%
equals	0.56s	0.04s	50017	50017	100%
bit-vector xor	1.09s	0.30s	2397869	2397869	100%
bit-vector or	0.92s	0.29s	2798115	2798115	100%
bit-vector and	0.78s	0.26s	2798778	2798778	100%
right shift	1.53s	0.67s	3199611	3199611	100%
left shift	1.58s	0.42s	3200933	3200933	100%
arithmetic shift	1.66s	1.26s	3249594	3249594	100%
addition	1.82s	0.29s	1137672	1689706	67%
multiplication	20.45s	1.47s	148531	162680	91%
multiplication (max $n = 6$)	47.03s	240.19s	148350	197310	75%
unsigned division	111.80s	7.04s	3041509	3057485	99%
unsigned remainder	120.01s	40.49s	763045	1122934	67%
signed division	59.78s	29.82s	1198292	3053684	39%
signed remainder	58.61s	136.00s	210381	841675	24%

Table 4.3c: Comparison of unit propagation and bit-blasting at 50%. 100000 iterations at 64 bits.

The unsigned division propagator fixed 99% of the possible bits. The signed modulus, remainder and division operations are the slowest; our implementation of these propagators is the least refined.

The clause encoding that the SAT solver uses is incremental, in that if some assignments change, only part of the work is redone. This contrasts to our propagator which begin again whenever an assignment changes. From Table 4.3d, at 95% known values our unsigned division propagator reaches fixed point 35 times faster than unit propagation does. So each unsigned division operation needs to be evaluated at least 35 times with various assignments, before the advantage of being incremental begins to outweigh the cost of having many clauses to propagate over. Because of this, propagator based approaches will be superior to CNF based approaches on easy problems which do not require the operation to be re-evaluated often.

To measure what percentage of the possible assignments our propagators derived, we ran the RSY algorithm on 1000 instances with various levels of information known. The results are in Table 4.4a – Table 4.4c. *Time* is the time to call both the propagator and RSY on the initial assignment. *Initial* is the number of bits initially randomly set in the instances, this varies with the percentage of values initially

Operation	95%				
	UP time	prop. time	UP bits	prop. bits	%
signed greater than equals	1.74s	0.32s	12597	13081	96%
unsigned less than	1.73s	0.26s	12554	13057	96%
equals	1.19s	0.03s	5067	5067	100%
bit-vector xor	1.34s	0.15s	866127	866127	100%
bit-vector or	1.10s	0.24s	599980	599980	100%
bit-vector and	1.17s	0.23s	599369	599369	100%
right shift	4.23s	0.55s	319839	319839	100%
left shift	4.14s	0.62s	320507	320507	100%
arithmetic shift	4.81s	1.00s	324578	324578	100%
addition	3.20s	0.26s	898103	907318	98%
multiplication	115.68s	10.46s	735701	929134	79%
multiplication (max n = 6)	150.77s	359.27s	736446	929917	79%
unsigned division	196.44s	5.83s	339424	340891	99%
unsigned remainder	205.60s	45.01s	694296	758022	91%
signed division	212.79s	14.28s	331093	346955	95%
signed remainder	218.18s	95.40s	644806	759474	84%

Table 4.3d: Comparison of unit propagation and bit-blasting at 95%. 100000 iterations at 64 bits.

assigned. *Prop* is the final (not the additional) number of values assigned after running our propagator. *Max* is the final number of values assigned after running RSY. *Found* is the percentage of possible additional bits fixed by our propagator.

These tables show that no initial assignment was discovered which caused the propagators we believe to be optimal, and the RSY algorithm to yield different assignments.

The tables also show why custom implementations of the propagators are necessary, versus using the RSY algorithm. For instance, 1,000 32-bit bit-vector exclusive-or propagations at 5% information took 2.92 seconds, this contrasts to the bit-vector exclusive-or propagator which took (Table 4.3b) 0.11 seconds to propagate on 100,000 64-bit assignments. That is, the custom implementation was approximately 5,000 times faster.

As the amount of known information increased, the RSY algorithm took less time. As the information increases, there are fewer possible assignments that the unassigned variables can take. This reduces the number of times the RSY algorithm needs to call the SAT solver.

To measure what percentage of the possible assignments our propagators derived at low bit-widths, we compare the results of the CNF encoding, and propa-

Operation	1%				
	Time	Initial	Prop.	Max	Found
signed greater than equals	2.48s	627	627	627	100%
unsigned less than	2.81s	639	639	639	100%
equals	2.05s	658	661	661	100%
bit-vector xor	3.12s	967	971	971	100%
bit-vector or	2.51s	1189	1495	1495	100%
bit-vector and	2.75s	1111	1424	1424	100%
right shift	7.18s	983	4687	4687	100%
left shift	7.08s	906	4648	4648	100%
arithmetic shift	7.37s	1027	1937	1937	100%
addition	5.77s	956	956	956	100%
multiplication	58.30s	911	922	922	100%
unsigned division	315.37s	967	3280	3285	99%
unsigned remainder	1092.83s	964	974	974	100%
signed division	159.61s	947	954	955	87%
signed remainder	313.00s	947	947	954	0%

Table 4.4a: Calculating the effect of the best propagator on 1000 32-bit operands; 1% of bits provided at random. 'Time' is the time to call both the propagator and RSY. 'Initial' is the number of bits initially randomly set in the instances. 'Prop' is the number of assignments after the propagator finished. 'Max' is the number of assignments after RSY finished. 'Found' is the percentage of possible bits fixed by our propagator.

gators versus the exhaustive approach (section 4.8) at small bit-widths. The results are shown in Table 4.5. The percentage gives the percentage of initial assignments where the exhaustive approach and the propagators or CNF respectively, did not produce the same answer. Unlike the prior tables, this does not count the extra assignments; if the propagators fixed 10 of 11 possible assignments then we count this as a failure. Unlike the prior tables we also generate conflicting assignments, for instance $(1 + 0) = 0$, if unit propagation does not report a conflict, or the propagator does not report a conflict. This again is considered a failure.

Again we see the propagators we expect to be optimal have a 0% failure rate.

Unit propagation on the CNF encoding of left and right shift is not optimal in 1.8% of cases for a bit-width of 5. This shows that the random assignments we produced for the prior tables over-estimated the power of unit propagation on the CNF.

Interestingly, Table 4.5 shows that at 5-bits the CNF with unit propagation has almost 4 times fewer missed assignments than our multiplication propagator. That is, it is considerably better on the lower order bits than the multiplication propagator

Operation	5%				
	Time	Initial	Prop.	Max	Found
signed greater than equals	2.51s	3231	3231	3231	100%
unsigned less than	2.50s	3240	3240	3240	100%
equals	1.90s	3219	3257	3257	100%
bit-vector xor	2.92s	4927	5003	5003	100%
bit-vector or	2.32s	5651	7147	7147	100%
bit-vector and	2.58s	5662	7192	7192	100%
right shift	5.57s	4797	19568	19568	100%
left shift	5.48s	4769	20507	20507	100%
arithmetic shift	5.91s	5223	18120	18120	100%
addition	5.49s	4836	4843	4843	100%
multiplication	67.55s	4782	4831	4831	100%
unsigned division	235.35s	4735	14182	14208	99%
unsigned remainder	908.75s	4685	4737	4759	70%
signed division	249.20s	4751	6167	6967	63%
signed remainder	576.09s	4895	4926	4971	40%

Table 4.4b: Calculating the effect of the best propagator on 1000 32-bit operands; 5% of bits provided at random.

Operation	50%				
	Time	Initial	Prop.	Max	Found
signed greater than equals	1.31s	32666	32773	32773	100%
unsigned less than	1.29s	32890	32981	32981	100%
equals	1.15s	32670	33152	33152	100%
bit-vector xor	1.24s	55748	59673	59673	100%
bit-vector or	1.19s	54143	62229	62229	100%
bit-vector and	1.23s	53981	61879	61879	100%
right shift	2.31s	47979	64072	64072	100%
left shift	2.34s	47955	63870	63870	100%
arithmetic shift	2.51s	48572	64493	64493	100%
addition	2.00s	53523	56292	56292	100%
multiplication	43.74s	48740	49471	50527	40%
unsigned division	52.98s	48352	62780	62864	99%
unsigned remainder	58.35s	50631	53064	56059	44%
signed division	80.45s	48597	62480	62656	98%
signed remainder	105.47s	50336	52402	55361	41%

Table 4.4c: Calculating the effect of the best propagator on 1000 32-bit operands; 50% of bits provided at random.

that we have described. Given the smaller number of missed assignments, this might a better basis for the approach we described (section 4.9) for building an optimal multiplication propagator.

Table 4.5 shows that unit propagation applied to STP2's 2-bit addition CNF is not maximally precise. After unit propagation, there are 16 assignments which are not maximally precise. These assignments, excluding those equivalent via commutativity, are:

$$(\langle 0\star\rangle + \langle 0\star\rangle) = \langle 00\rangle$$
$$(\langle 0\star\rangle + \langle \star\star\rangle) = \langle 01\rangle$$
$$(\langle 0\star\rangle + \langle 0\star\rangle) = \langle \star 1\rangle$$
$$(\langle 0\star\rangle + \langle \star\star\rangle) = \langle 11\rangle$$
$$(\langle 0\star\rangle + \langle 1\star\rangle) = \langle \star 1\rangle$$
$$(\langle 0\star\rangle + \langle 1\star\rangle) = \langle 10\rangle$$
$$(\langle \star\star\rangle + \langle 1\star\rangle) = \langle 01\rangle$$
$$(\langle 1\star\rangle + \langle 0\star\rangle) = \langle \star 1\rangle$$
$$(\langle 1\star\rangle + \langle 0\star\rangle) = \langle 10\rangle$$
$$(\langle 1\star\rangle + \langle 1\star\rangle) = \langle 00\rangle$$
$$(\langle \star\star\rangle + \langle 1\star\rangle) = \langle 11\rangle$$
$$(\langle 1\star\rangle + \langle 1\star\rangle) = \langle \star 1\rangle$$

In each of the assignments above, the carry is known and can be used to deduce additional bits. Note, these assignments are particular to STP2. For instance, Minisat+ [ES06] creates a CNF for addition operations which is maximally precise under unit propagation.

4.11 Related Work

Automatically generating propagators avoids the effort of building efficient propagators. It is practical to automatically generate propagators for simple operations, in particular the bit-wise operations (e.g. bit-vector exclusive-or, or bit-vector and). However, the propagators for more complex operations like multiplication and division that are automatically derived are currently too slow to be useful.

Operation	1 bits		2 bits		3 bits		4 bits		5 bits	
	Prop	BB	Prop	BB	Prop	BB	Prop	BB	Prop	BB
unsigned less than	0.0	0.0	0.0	1.2	0.0	3.4	0.0	5.4	0.0	6.7
equals	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bit-vector xor	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bit-vector or	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bit-vector and	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
right shift	0.0	0.0	0.0	1.1	0.0	2.0	0.0	2.1	0.0	1.8
left shift	0.0	0.0	0.0	1.1	0.0	2.0	0.0	2.1	0.0	1.8
addition	0.0	0.0	0.0	2.2	0.0	5.2	0.0	7.9	0.0	10.0
subtraction	0.0	0.0	0.0	2.2	0.0	5.2	0.0	7.9	0.0	10.0
multiplication	0.0	0.0	0.1	0.4	3.3	1.0	7.0	1.9	11.1	2.8
unsigned division	40.7	0.0	22.1	1.5	13.8	3.8	10.3	3.8	8.2	4.0
unsigned remainder	22.2	7.4	28.8	16.5	27.7	20.3	25.3	23.4	23.5	25.3

Table 4.5: Percentage of all the possible initial assignments at different bit-widths, where the CNF encoding with unit propagation, or our propagators either, missed at least one possible assignment, or missed a conflicting assignment.

Regehr and Duongsaa [RD06] automatically derive propagators for bit-vector operations on **3**. The propagators transfer information from operands to results and not vice-versa. Their automatically generated bit-vector xor propagator performs 875,000 32-bit operations per second. However, the multiplication propagator performs only 400 32-bit operations per second. We follow them in testing our propagators by exhaustively taking the join of concrete elements. Regehr and Reid [RR04] also automatically derived **3** propagators, but for small bit-widths.

Bardin et al. [BHP10], describe a bit-vector solver with propagators for **3** (which they call the BitList domain). If we combined the ability to search and backtrack with the **3** propagators that we describe, it would also be a decision procedure. For arithmetic constraints they focus on cheap and correct propagators, as distinct from our work which focuses on precise propagators.

Michel and Van Hentenryck [MV12] give maximally precise propagators for an equivalent domain to **3**. They give algorithms for the bitwise operations, the comparisons, shifting and addition. They focus on bit-vectors which are shorter than the machine's bit-width, so can efficiently be implemented using data parallel machine instructions. They do not investigate multiplication and allied operations, or give experimental results.

Naveh et al. [NRJ+07] combine several domains including **3**, which they call “set of masks”, in a constraint propagation solver. Their paper gives only high level information about the propagators used.

Achterberg [Ach07] uses exclusive-or normal form to propagate both ways (between the operands and result) of bit-wise multiplication. Because a representation in exclusive-or normal form can grow exponentially large, a limit is placed on the size of formulae. This is an alternative approach to our work in section 4.9 for setting the least significant $\star$ bits involved in a multiplication.

Budiu et al.[BSWG00, BG00] use bit propagation, propagating from operands to result, to reduce the size of circuits generated by a compiler for reconfigurable hardware. The technical report gives pseudo-code for the propagators, which propagate other information, but not bit propagation information, from result to operands.

Strided intervals limit the values between the upper and lower bound to also have a fixed number of trailing bits set to zero. Balakrishnan [Bal07] describes strided-interval propagators for some bit-vector operations. Like a bitwise analysis, strided intervals can determine that all the trailing bits are zero.

Jung et al. [JBKW08] convert a type of graph that subsumes BDDs, to maximally precise CNF clauses. This is an alternate approach for achieving a maximally precise CNF encoding of operations like addition.

4.12 Conclusion

We described a theory-level analysis to determine the assignment some variables must take. We focused on building precise propagators, in particular we described the implementation of the equality, addition, multiplication and unsigned division propagators.

Our results show that our propagators can propagate information that unit propagation over the CNF does not. Using the propagators resulted in about 10% fewer failures on the test problems we chose. This shows that useful information is “lost in translation” to CNF. Reasoning at the theory-level may avoid the need to encode some operations to CNF, which for large bit-widths may overwhelm the SAT solver.

Using RSY to produce the effect of propagators allowed us to measure whether bit propagation simplified problems before investing the effort to build efficient implementations of the propagators. We believe this is an under-appreciated advantage of RSY. By solving a problem with bit propagation, using RSY, and then subtracting the time spent in RSY, we measured the maximum possible speed up from applying bit propagation, assuming propagators took zero time. By comparing this to the time taken to solve the problem without bit propagation, the maximum possible speedup can be measured.

The difficulty in applying RSY is that our implementation is slow for some operations. Table 4.4a – Table 4.4c shows the time taken to use RSY on 32-bit values, compared to the results in Table 4.3a – Table 4.3d. For instance, our implementation of RSY propagates arithmetic left shift more than 180 times slower than our hand crafted shift propagator.

The C++ source code for our implementation is included in the publicly available STP source code repository.

After analysis, some rewrites may be possible. For example, given ($t_0 \geq_s t_1$), where the top bit of both t_0 and t_1 is fixed, the signed comparison can be replaced by a cheaper unsigned comparison. We do not perform such strength reductions.

Adding redundant clauses to improve unit propagation on CNF encodings is possible; Eén and Sörensson [ES06] use redundant clauses to improve unit propagation of their bit-vector addition encoding. The CNF encoding of some operations could be improved by such clauses. The most promising being addition, which has a simple, repeating structure. If extra redundant clauses were added, then unit propagation on the CNF would improve, reducing an advantage of the propagator based approach. Unfortunately, adding such redundant clauses into the bit-blasted (section 3.8) AIG representation can be difficult. The AIGs are good at eliminating such redundancy.

The preference for propagators built for constraint solving is for them to be cheap and correct, rather than precise. Because in bit propagation the propagators are only run twice to global fixed point, we knew that the extra runtime cost of precise propagators was not onerous. However, we believe it is not widely appreciated how straightforward it is to measure a propagator's precision for small domain sizes, as we have done. Unless optimal propagators are first built, then measured, it cannot be conclusively decided whether or not the extra time spent ensuring maximal precision is justified.

In this chapter we showed that:

- applying bit propagation as a pre-processor speeds up solving bit-vector problems;
- building optimal **3** propagators for many operations is practical; and
- measuring the precision of propagators on small domain sizes is a good way to test that they are precise.

Propagators are promising for bit-vector solving because they can express the same logic more compactly than is possible in CNF. In particular, improving the amount of propagation of CNF based representations might require impractically many redundant clauses to be added. Propagators consume less memory because their reasoning does not need to be entirely statically expressed in CNF.

5 Building a Better Array Solver

IN the previous chapters, we described how to build a better bit-vector solver. In this chapter we describe how to build a better array solver. STP2 is an open-source solver for a fragment (without array extensionality) of the `QF_ABV` language, a combined quantifier-free theory of bit-vectors and arrays with extensionality.

Efficiently solving array problems is important for software verification. STP2 did not compete in the `QF_ABV` division of the SMT-COMP 2011 because it does not implement array extensionality, but it is competitive for `QF_ABV` problems without extensionality (see subsection 5.6.2).

In this chapter we compare approaches to enforcing the *function congruence constraint* ($\mathcal{FCC}$). The $\mathcal{FCC}$ ensures that a relation is in fact a function, that is,

$$\forall(i, j) : (i = j) \implies (f(i) = f(j))$$

We consider the $\mathcal{FCC}$ just for unary functions. An *instance* of the $\mathcal{FCC}$ enforces the $\mathcal{FCC}$ for two particular applications of the same function. Because STP2 does not handle quantifiers, the $\mathcal{FCC}$ is instantiated for each pair of function applications. If there are l applications of function f, then in the worst case the $\mathcal{FCC}$ is instantiated for each distinct pair of function applications, that is, $\frac{l(l-1)}{2}$ times. We refer to the exhaustive approach of asserting all $O(l^2)$ of these $\mathcal{FCC}$ instances as *Ackermannization* ($\mathcal{A}ck$) ([Ack54] cited by [dMB08a]).

To avoid necessarily asserting quadratically many $\mathcal{FCC}$ instances via $\mathcal{A}ck$, we use the *counter-example guided abstraction-refinement* approach popularised by Clarke et al. [CGJ+00], we refer to the approach as $\mathcal{A}bsref$. $\mathcal{A}bsref$ omits the $\mathcal{FCC}$ when the problem is initially asserted to the SAT solver. If the SAT solver returns a candidate model, then, and only then, outside the SAT solver, is the $\mathcal{FCC}$ checked. If unsatisfied $\mathcal{FCC}$ instances exist, then they are asserted to the SAT solver, otherwise the problem is satisfiable and work stops. Iterating between checking the $\mathcal{FCC}$ and SAT solving continues until either the SAT solver generates a model that satisfies the $\mathcal{FCC}$ (even though all the $\mathcal{FCC}$ instances might not be asserted), or until the SAT solver establishes that the problem is unsatisfiable. Leaving out the $\mathcal{FCC}$ is *abstraction*, checking the $\mathcal{FCC}$ and iteratively adding in unsatisfied $\mathcal{FCC}$ instances is *refinement*. The problem is initially under-constrained, then iteratively refined until, in the worst case, it is equisatisfiable with the result of $\mathcal{A}ck$.

Another approach, that we refer to as *Delayed Congruence Instantiation* ($\mathcal{DCI}$) builds the ability to generate $\mathcal{FCC}$ instances into the SAT solver. This approach avoids the expense of the $\mathcal{A}bsref$ refinement loop, because enforcing the $\mathcal{FCC}$ is performed inside the SAT solver. It is tailored to assert instances of the $\mathcal{FCC}$ close to when they have an effect. There is no widely used name for this type of approach, we were introduced to it as “Lazy Clause Generation” [OSC09]. We give the details of our implementation in section 5.5. In short, whenever the SAT solver makes completely the same assignment to the operands of two function applications, an instance of the $\mathcal{FCC}$ for those two function applications is asserted without leaving the SAT solver.

We operate on first-order quantifier-free formulae with single-dimensional bit-vector arrays without extensionality. Both the indices and the values stored in the arrays are fixed-width bit-vectors, that is, they have finite width. It is not possible to use either arrays or propositions as indices or values. An array is a total function from bit-vectors to bit-vectors of a potentially different bit-width.

Let a be a variable ranging over arrays. We use the bit-widths of arrays as superscripts on the variables name, for instance $a_\ell^{[2:3]}$ maps from four 2-bit vectors to 3-bit vectors. In general, we do not distinguish between array literals, and array terms. When it is necessary, we mark array literals with a subscript ℓ, for example a_ℓ. The functions added for the array theory are:

- *Select* returns the bit-vector stored at an index; that is, *select*$(a^{[n:m]}, t^{[n]})$ returns the value from $a^{[n:m]}$ at index $t^{[n]}$.

- *Store* non-destructively creates a new array that replaces the value at an index; $store(a^{[n:m]}, t_0^{[n]}, t_1^{[m]})$ creates a new array the same as $a^{[n:m]}$ except that at index $t_0^{[n]}$ the value $t_1^{[m]}$ is stored.

  More formally the axiom introduced is:

$$\forall(a, i, j, e)(select(store(a, i, e), j) = ite(i = j, e, select(a, j))) \tag{5.1}$$

  This is the "forwarding property" of hardware verification research: that a *select* returns the value most recently stored at an index.

- An array if-then-else (ITE) $ite(p, a_0^{[n:m]}, a_1^{[n:m]})$ returns $a_0^{[n:m]}$ if p is 1, and $a_1^{[n:m]}$ otherwise.

Select is sometimes called "read", and *store* is sometimes called "write". *Store* and array-ITE can be used to create nested array terms.

The `QF_ABV` theory also includes the extensionality axiom, which states that two arrays are equal if they hold the same value at each index:

$$(a_0^{[n:m]} = a_1^{[n:m]}) \Leftrightarrow \forall(i)(select(a_0^{[n:m]}, i) = select(a_1^{[n:m]}, i)) \tag{5.2}$$

However, we do not allow equality or disequality to be applied to terms of array type. Some array problems with extensionality can be rewritten to equisatisfiable array problems without extensionality by using:

$$(a_0^{[n:m]} \neq a_1^{[n:m]}) \implies \exists(i)(select(a_0^{[n:m]}, i) \neq select(a_1^{[n:m]}, i))$$

Software verification problems often model memory as a single array with 2^{32} or 2^{64} possible indices. So, to be practical, a solver must use memory or time which is sub-linear in the number of possible indices.

We use three phases to solve combined bit-vector and array problems. First, the array part of the problem is simplified by applying rewrite rules. Second, array-ITE and *store* expressions are removed. Third, the $\mathcal{FCC}$ instances corresponding to the remaining *select* expressions are instantiated by one of three approaches to be discussed below.

Of the three approaches, the $\mathcal{A}ck$ approach that we present is well known. The $\mathcal{A}bsref$ approach that we describe was implemented in STP 0.1; we provide a more complete description of the algorithm than has previously been published, but we did not develop the approach. The $\mathcal{DCI}$ approach that we present is novel.

5.1 Simplifying

STP2 applies the following rewrite rules to simplify expressions containing arrays. These simplifications are applied when expressions are created, what we call at creation-time (section 3.3). The variables below are typed term variables which match arbitrary expressions, so that distinct variable names can match distinct syntactic expressions. p is an arbitrary expression of propositional type:

$$\begin{aligned}
ite(1, a_0^{[n:m]}, a_1^{[n:m]}) &\rhd a_0^{[n:m]} \\
ite(0, a_0^{[n:m]}, a_1^{[n:m]}) &\rhd a_1^{[n:m]} \\
ite(p, a_0^{[n:m]}, a_0^{[n:m]}) &\rhd a_0^{[n:m]} \\
ite(p, a_0^{[n:m]}, ite(p, a_1^{[n:m]}, a_2^{[n:m]})) &\rhd ite(p, a_0^{[n:m]}, a_2^{[n:m]}) \\
ite(p, ite(p, a_0^{[n:m]}, a_1^{[n:m]}), a_2^{[n:m]}) &\rhd ite(p, a_0^{[n:m]}, a_2^{[n:m]}) \\
ite(not(p), a_0^{[n:m]}, a_1^{[n:m]}) &\rhd ite(p, a_1^{[n:m]}, a_0^{[n:m]})
\end{aligned}$$

$$\begin{aligned}
store(store(a_0^{[n:m]}, i^{[n]}, j^{[m]}), i^{[n]}, k^{[m]}) &\rhd store(a_0^{[n:m]}, i^{[n]}, k^{[m]}) \\
store(a_0^{[n:m]}, t_0^{[n]}, select(a_0^{[n:m]}, t_0^{[n]})) &\rhd a_0^{[n:m]}
\end{aligned}$$

$$select(store(a_0^{[n:m]}, t_0^{[n]}, t_1^{[m]}), t_0^{[n]}) \rhd t_1^{[m]}$$

$$\begin{aligned}
select(store(a_0^{[n:m]}, t_0^{[n]}, t_1^{[m]}), t_2^{[n]}) &\rhd select(a_0^{[n:m]}, t_2^{[n]}), \\
&\quad \text{when } t_0^{[n]} = t_2^{[n]} \text{ is equivalent to } 0
\end{aligned}$$

The last rule is the most complicated. It is a conditional rewrite that checks if the index of a *select* and the index of a *store* must be different; if so the *store* is discarded. This rule applies often, especially when the indices of the *select* and the *store* are different constants.

Even for an expression with sharing, these rules do not increase the total number of expressions. They are what we call sharing-aware (section 2.7). If there are already l distinct expressions, after applying these rewrite rules there will be l or fewer expressions.

STP2 also replaces *select* expressions with both a constant index and a constant *result* through out the problem. So if $select(a_\ell, c_0) = c_1$ is conjoined with the root node, and c_0 and c_1 are both constants, then $select(a_\ell, c_0)$ is replaced by c_1 throughout the problem.

5.2 Removing Store and Array-ITEs

Without extensionality, it is straightforward to remove *select* and array-ITE terms by repeatedly applying Equation 5.3 and Equation 5.4 below to a fixed point. Applying these rewrite rules generates equivalent expressions that contain the array theory axiom (Equation 5.1). After applying these rules the only array terms remaining are *select* terms. This considerably simplifies the later algorithms; the *select* terms require just the $\mathcal{F}CC$ to be enforced between them. Applying the equations gives a reduction from the theory of arrays to the theory of uninterpreted functions. Later, when converting to CNF, we further reduce from the theory of uninterpreted functions and bit-vectors to propositional logic.

ITE-lifting removes array-ITEs by converting them to term-ITEs:

$$select(ite(p, a_0^{[n:m]}, a_1^{[n:m]}), t^{[n]}) \triangleright ite(p, select(a_0^{[n:m]}, t^{[n]}), select(a_1^{[n:m]}, t^{[n]})) \tag{5.3}$$

Select-over-store elimination, which is Equation 5.1 expressed as a rewrite rule, removes the *store* function:

$$select(store(a^{[n:m]}, t_0^{[n]}, t_2^{[m]}), t_1^{[n]}) \triangleright ite(t_0^{[n]} = t_1^{[n]}, t_2^{[m]}, select(a^{[n:m]}, t_1^{[n]})) \tag{5.4}$$

A disadvantage of this approach is that, in the worst case, it will introduce a quadratic number of extra expressions. Each application of Equation 5.4 may create a new, unique, term-ITE. Given an expression with sharing, several *select* expressions can reference either the same ITE or *store* term. So, when Equation 5.4 is applied it will create a distinct equality term between the read index of the *select* term and write index of the *store* term. Roughly, given l *select* terms that share a reference to a chain of m *store* terms, after applying Equation 5.4 to a fixed point, $l \times m$ ITE expressions are introduced. We investigate this quadratic blow-up in subsection 5.6.4.

Applying Equation 5.3 to shared expressions has exponential time complexity if caching is not performed. The number of paths through array-ITEs can grow exponentially in the worst case.

Example 5.1

As an example of the worst-case behaviour, consider the following conjuncts, where S_0 is the root node, and S_1 to S_4 are syntactic variables:

$$\begin{aligned}
S_0 &= ite(p_0, S_1, S_2) \\
S_1 &= ite(p_1, S_2, S_3) \\
S_2 &= ite(p_2, S_3, S_4) \\
S_3 &= ite(p_3, S_4, a_0^{[n:m]}) \\
S_4 &= ite(p_4, a_0^{[n:m]}, a_1^{[n:m]})
\end{aligned}$$

There are 8 distinct paths to reach $a_0^{[n:m]}$. If the same structure is extended to S_k, then the number of distinct paths to reach $a_0^{[n:m]}$ equals the k^{th} Fibonacci number. ■

Our implementation performs caching to avoid repeated rewrites of the same expression. If an array-ITE has l *select* expressions as ancestors, then the array-ITE will be rewritten by Equation 5.3 at most l times.

In the worst case, the total number of applications of Equation 5.3 and Equation 5.4 is quadratic. It is bounded by the total number of *select* expressions multiplied by the total number of expressions.

Example 5.2

Applying Equation 5.3 to the expression:

$$select(ite(b_0, ite(b_1, a_0, a_1), a_1), t)$$

gives:

$$ite(b_0, ite(b_1, select(a_0, t), select(a_1, t)), select(a_1, t))$$

The initial expression has two references to the shared array-term a_1. Note, likewise, the rewritten expression has two references to the term $select(a_1, t)$. The result of applying Equation 5.3 has similar sharing to the initial expression. ■

5.3 Eliminating *selects*: Ackermannization

In this section we describe two implementations of $\mathcal{A}ck$. Before applying $\mathcal{A}ck$, the array-ITEs and *stores* are removed, as described in the previous section. $\mathcal{A}ck$ eliminates *select* terms by instantiating the $\mathcal{FCC}$, that is it reduces to the theory of uninterpreted functions by writing the $\mathcal{FCC}$ into the problem.

If there are l *selects* with syntactically distinct index terms $t_0^{[n]} \ldots t_{l-1}^{[n]}$ from array $a_\ell^{[n:m]}$, then $\mathcal{A}ck$ creates all $\frac{l(l-1)}{2}$ $\mathcal{FCC}$ instances:

$$\forall_{0 \le j < k < l}((t_j^{[n]} = t_k^{[n]}) \implies (select(a_\ell^{[n:m]}, t_j^{[n]}) = select(a_\ell^{[n:m]}, t_k^{[n]})))$$

Example 5.3

If an expression contains 5 *select* expressions accessing array a_0, and 10 *select* expressions accessing array a_1, then as many as $\frac{5(5-1)}{2} + \frac{10(10-1)}{2} = 55$ $\mathcal{FCC}$ instances are asserted. ■

The first implementation that we present, which we call $\mathcal{A}ck_{cnf}$, asserts the $\mathcal{FCC}$ directly to the SAT solver (Algorithm 5.1). $\mathcal{A}ck_{cnf}$ creates the $\mathcal{FCC}$ instances directly as CNF, bypassing the expensive bit-blasting and AIG to CNF encoding steps. Note that line 3 traverses the *select* terms after they have been topologically sorted, so the index of the *select* contains no *select* terms.

We use the $\mathcal{FCC}$ instance CNF representation introduced by Biere and Brummayer [BB08a]. The representation was described in more detail in Section 2.11.6 of Robert Brummayer's PhD thesis [Bru09]. Algorithm 5.2 shows how to generate the CNF. For two *selects* of index bit-width n and result bit-width m, the algorithm adds $1 + 2n + 2m$ clauses and creates $n + 1$ fresh variables. Note that the clauses allow e to

Algorithm 5.1 $\mathcal{A}ck_{cnf}$ removes *select* terms by asserting $\mathcal{FCC}$ instances directly to the SAT solver. The *subterms* method returns a list of e's subterms topologically sorted.

```
Require: $e$ a formula
 1: Create $selects$, a set of tuples of an array literal, an index, and a result
 2: $selects \leftarrow \{\}$
 3: for all $select(a, i) \in subterms(e)$ do                 // in topological order
 4:     Replace $select(a, i)$ in $e$ by a fresh variable $v$
 5:     Add $\langle a, i, v\rangle$ to $selects$
 6: end for
 7: Assert $e$ to the SAT solver          // All select terms have been removed from $e$
 8: for all distinct array literals $a \in selects$ do
 9:     Create $pairs$, a list of $\langle index, result\rangle$ pairs
10:     $pairs \leftarrow selects[a]$                       // Get all the selects of array $a$
11:     for all $j$ from $(1 \dots (size(pairs) - 1))$ do
12:         for all $k$ from $(0 \dots (j - 1))$ do
13:             if $(pairs[j].index = pairs[k].index)$ does not equal 0 after bit-vector
    theory-level simplifications then
14:                 FCC_INSTANCE$(pairs[j], pairs[k])$                 // Algorithm 5.2
15:             end if
16:         end for
17:     end for
18: end for
```

Algorithm 5.2 Creating $\mathcal{FCC}$ instances in CNF, after Biere and Brummayer [BB08a, Bru09].

```
Require: $i_0^{[n]}, w_0^{[m]}$          // The index and result corresponding to a select
Require: $i_1^{[n]}, w_1^{[m]}$       // The index and result corresponding to another select
 1: procedure FCC_INSTANCE$((i_0, w_0), (i_1, w_1))$
 2:     Create the fresh variables $v_0 \dots v_{n-1}, e$
 3:     for all $i$ from $0 \dots (n - 1)$ do
 4:         Output $(i_0[i] \wedge i_1[i]) \implies v_i$
 5:         Output $(\neg i_0[i] \wedge \neg i_1[i]) \implies v_i$
 6:     end for
 7:     Output $(v_0 \wedge \dots \wedge v_{n-1}) \implies e$        // Note: $\mu(e)$ is 1 when $\mu(i_0) = \mu(i_1)$
 8:     for all $i$ from $0 \dots (m - 1)$ do
 9:         Output $(e \wedge w_0[i]) \implies w_1[i]$
10:         Output $(e \wedge w_1[i]) \implies w_0[i]$
11:     end for
12: end procedure
```

be 1 when the indices are different. We have use the implication connective ($\implies$) for readability: note that all formulae output are disjunctions of literals.

Example 5.4

Consider applying Algorithm 5.2 to assert an $\mathcal{FCC}$ instance for $select(a^{[2:3]}, t_0)$, and $select(a^{[2:3]}, t_1)$, where the *selects* have been replaced by fresh variables w_0, w_1 respectively. Clauses are asserted to express that the indices' bits are pairwise equal. That is, these clauses are asserted:

$$\begin{aligned}
(t_0[0] \wedge t_1[0]) &\implies v_0\\
(\neg t_0[0] \wedge \neg t_1[0]) &\implies v_0\\
(t_0[1] \wedge t_1[1]) &\implies v_1\\
(\neg t_0[1] \wedge \neg t_1[1]) &\implies v_1
\end{aligned}$$

If the indices are pairwise equal then the clause ($v_0 \wedge v_1 \implies e$), forces e to be 1. When e is 1, the clauses:

$$\begin{aligned}
e \wedge w_0[0] &\implies w_1[0]\\
e \wedge w_1[0] &\implies w_0[0]\\
e \wedge w_0[1] &\implies w_1[1]\\
e \wedge w_1[1] &\implies w_0[1]\\
e \wedge w_0[2] &\implies w_1[2]\\
e \wedge w_1[2] &\implies w_0[2]
\end{aligned}$$

enforce that the bit-vectors $w_0^{[3]}$ and $w_1^{[3]}$ are the same bitwise. ■

Example 5.5

Consider some expression that contains three select sub-expressions: $select(a, 5)$, $select(b, t_0)$, and $select(a, select(b, t_0))$.

$\mathcal{A}ck_{cnf}$ (Algorithm 5.1) replaces $select(a, 5)$ by v_0, $select(b, t_0)$ by v_1, and $select(a, v_1)$ by v_2. Because there is a single *select* from b, no $\mathcal{FCC}$ instances are needed. The $\mathcal{FCC}$ instance for a is: $(5 = v_1) \implies (v_0 = v_2)$, which is encoded via Algorithm 5.2. ■

The $\mathcal{A}ck_{cnf}$ implementation creates a bit-vector equality between index terms (Algorithm 5.1 line 13), then simplifies it. The simplifications are simple rewrite rules that may simplify the equality to 0. If two indices are definitely not equal (the equality simplifies to 0), then no $\mathcal{FCC}$ instance is asserted. This omits the $\mathcal{FCC}$ instance in cases where indices are obviously not equal, for instance: $(t + 4 = t)$, $(8 = 10)$, and $((1^{[1]} :: t) = (0^{[1]} :: t))$. We assume a DAG expression as input, so the $\mathcal{A}ck$ algorithms have no need to optimise for encountering syntactically identical *select* terms.

An alternative method is $\mathcal{A}ck_{ite}$. It encodes the $\mathcal{FCC}$ as bit-vector terms using ITE expressions. It is given in Algorithm 5.3, and is identical to the $\mathcal{A}ck$ implementation of STP 0.1. The same method was also presented by Manolios et al. [MSV06]. The effect of $\mathcal{A}ck_{ite}$ is to remove array terms entirely from the problem, reducing the problem to a bit-vector problem.

Algorithm 5.3 $\mathcal{A}ck_{ite}$ which removes *select* terms by replacing them with term-ITEs.

```
Require: $e$, a formula
 1: Create $pairs$, a map from array literals to lists of pairs of bit-vector indices and
    results
 2: $pairs \leftarrow \{\}$
 3: for all $select(a, i) \in subterms(e)$ do                // in topological order
 4:     Create $t$, an expression
 5:     $t \leftarrow$ a fresh variable $v$
 6:     for all $j$ from $(0 \ldots (size(pairs[a]) - 1))$ do
 7:         $t \leftarrow ITE(i = pairs[a][j].index, pairs[a][j].result, t)$
 8:     end for
 9:     Replace $select(a, i)$ in $e$ by $t$
10:     Add the pair $(i, v)$ to the front of the list $pairs[a]$
11: end for
12: Output $e$
```

Example 5.6

Consider applying $\mathcal{A}ck_{ite}$ (Algorithm 5.3), given the *selects*: $select(a, s)$, $select(a, t)$, and $select(a, u)$. The first *select* is replaced by a fresh variable v_0, the second *select* is replaced by $ite(t = s, v_0, v_1)$, and the third *select* is replaced by $ite(u = s, v_0, ite(u = t, v_1, v_2))$. Note that $\mathcal{A}ck_{ite}$ must generate the term-ITE in a particular order; replacing $select(a, u)$ with $ite(u = t, v_1, ite(u = s, v_0, v_2))$ would be incorrect. ■

A major advantage of $\mathcal{A}ck$ is simplicity—it generates a single CNF representation of the problem. Because the SAT solver's programmatic interface is not needed, it is easy to change between SAT solvers. A consequence is that it is easy to upgrade to whichever is the best sequential or parallel SAT solver. Another advantage is that global AIG or CNF simplifications can be applied to the CNF. The next two approaches we investigate interface more closely with the SAT solver.

5.4 Eliminating *selects*: Abstraction-Refinement

Software verification problems may have a thousand or more *selects* from an array. With $\mathcal{A}ck$ this necessitates the inclusion of about five hundred thousand $\mathcal{F}CC$ instances. However, some $\mathcal{F}CC$ instances are unnecessary if:

- they constrain indices that because of other constraints can never be equal,
- they constrain *results* that because of other constraints can never be different, or,
- the satisfiability of the problem does not depend on the $\mathcal{F}CC$, for instance, if the bit-vector part of the problem alone is unsatisfiable.

$\mathcal{A}bsref$ overcomes the main problem of $\mathcal{A}ck$—that all the instances of the $\mathcal{F}CC$ are always sent to the SAT solver—by adding $\mathcal{F}CC$ instances as needed. In the worst case, all the $\mathcal{F}CC$ instances are asserted, so $\mathcal{A}bsref$ produces a formula equi-satisfiable with that from $\mathcal{A}ck$. In the best case, when the satisfiability is determined just by the bit-vector part of the problem, no $\mathcal{F}CC$ instances are asserted.

$\mathcal{A}bsref$ asserts an over-approximation of the problem to the SAT solver, then uses the SAT solver's models to determine which $\mathcal{F}CC$ instances to assert. Initially $\mathcal{A}bsref$ omits the $\mathcal{F}CC$. It lets the SAT solver generate a candidate model, and then checks whether that model satisfies the $\mathcal{F}CC$. $\mathcal{F}CC$ instances that are violated are asserted to the SAT solver. If the resulting formula is unsatisfiable, then work has finished. However, if it is satisfiable, the omitted $\mathcal{F}CC$ instances are checked, unsatisfied instances are asserted, and the SAT solver restarted. Because the SAT solver is solving an increasingly more constrained problem, an important practical consideration is that much of the SAT solver's state can be kept between invocations.

In the worst case, if a single $\mathcal{FCC}$ instance is asserted in each refinement iteration, there will be quadratically many iterations. Each refinement step calls the SAT solver, which has a startup cost. To reduce the worst case, $\mathcal{FCC}$ instances can be asserted even if they currently evaluate to 1. The $\mathcal{A}bsref$ algorithm (Algorithm 5.4) of STP has no more refinement iterations than there are distinct *select* expressions. The $\mathcal{A}bsref$ algorithm of STP2 differs from that of STP 0.1 in that, like $\mathcal{A}ck_{cnf}$, it asserts $\mathcal{FCC}$ instances as CNF using Algorithm 5.2. So the majority of the treatment in Vijay Ganesh's PhD thesis [Gan07] Section 4.5 is still accurate.

Example 5.7

If one array literal appears in 30 *selects*, and the other in 100 *selects*, then there will be at most 130 refinement iterations. However, if instead $\mathcal{FCC}$ instances were added singly, there are $\frac{100\times 99}{2} + \frac{30\times 29}{2} = 5385$ possible refinement iterations. ■

Implementations of $\mathcal{A}bsref$ make a trade-off between calling the SAT solver many times and asserting many unnecessary $\mathcal{FCC}$ instances, that is, between asserting unnecessary clauses like $\mathcal{A}ck$ does, and risking quadratically many refinement iterations if single unsatisfied $\mathcal{FCC}$ instances are asserted. Because STP2 asserts extra $\mathcal{FCC}$ instances, for some problems in our suite (see subsection 5.6.3) it is more than two thousand times faster than Boolector, another $\mathcal{A}bsref$ based solver. However, the risk is that the redundant $\mathcal{FCC}$ instances increase the memory required and slow down SAT solving.

It is common in software verification problems for the values at some indices to be specified. The CNF encoding of an equality between a constant and a variable is smaller than the encoding between two variables. The $\mathcal{A}bsref$ implementation sorts the list of *selects* so that constant indices are checked first. Sorting the indices does not matter to the $\mathcal{A}ck$ approach because the same number of $\mathcal{FCC}$ instances will be asserted regardless of sorting. By sorting the list of *selects* we hope for $\mathcal{A}bsref$ to assert fewer CNF clauses overall.

Example 5.8

Consider applying Algorithm 5.4 to the expression $((select(a, 5) = select(a, t_0)) \wedge (select(a, t_1) = t_2))$.

Algorithm 5.4 STP2's algorithm for $\mathcal{A}bsref$

```
Require: $e$, a formula
 1: Create $original$
 2: $original \leftarrow e$
 3: Create $pairs$, a map from array literals to lists of pairs of bit-vector indices and
    results
 4: $pairs \leftarrow \{\}$
 5: for all $select(a, i) \in subterms(e)$ do                  // in topological order
 6:     Replace $select(a, i)$ in $e$ by a fresh variable $v$
 7:     Add the pair $(i, v)$ to the list $pairs[a]$
 8: end for
 9: Assert $e$ to the SAT solver
10: if $SAT\_solve()$ is unsatisfiable then
11:     return unsatisfiable
12: end if
13: Create: $next$, $later$, lists of CNF clauses
14: for all distinct array literals $a$ encountered do
15:     Sort the indices in $pairs[a]$, so that constant indices are first.
16:     for all $j$ from $(0 \ldots (size(pairs[a]) - 1))$ do
17:         for all $k$ from $(j + 1 \ldots (size(pairs[a] - 1)))$ do
18:             if $(\mu(pairs[a][j].index) = \mu(pairs[a][k].index)) \wedge (\mu(pairs[a][j].result) \neq$
    $\mu(pairs[a][k].result))$ then
19:                 $next.push(\text{FCC\_INSTANCE}(pairs[a][j], pairs[a][k]))$
20:             else
21:                 $later.push(\text{FCC\_INSTANCE}(pairs[a][j], pairs[a][k]))$
22:             end if
23:         end for
24:         if $size(next) > 0$ then
25:             Assert $next$ to the SAT solver
26:             Empty $next$
27:             if $SAT\_solve()$ is unsatisfiable then
28:                 return unsatisfiable
29:             else if $\mu(original) = 1$ then
30:                 return satisfiable
31:             end if
32:         end if
33:     end for
34: end for
35: if $size(later) > 0$ then
36:     Assert $later$                                        // Equisatisfiable to $\mathcal{A}ck$.
37:     return $SAT\_solve()$
38: end if
```

Each select expression is replaced by a fresh variable, giving $(v_0 = v_1) \wedge (v_2 = t_2)$, which is asserted to the SAT solver. The original formula will be evaluated with the assignment from the SAT solver.

Consider a model: $\mu(v_0) = \mu(v_1) = 5, \mu(v_2) = \mu(t_2) = 6, \mu(t_0) = \mu(t_1) = 2$. The original formula is evaluated with this assignment. To ensure that the $\mathcal{FCC}$ applies, the result at every index is kept when it is encountered. Because v_0 is 5, $select(a, 5) = 5$ is stored. Because t_0 is 2, $select(a, 2) = 5$. Now $v_2 = 6$, and $t_1 = 2$, but $select(a, 2)$ has already been set to 5, so the prior value is used. The *original* formula evaluates to $(5 = 5) \wedge (5 = 6)$, which is 0.

Refinement is now applied to each distinct pair of select indices. We have $(5 \neq \mu(t_0))$, so the instance $((5 = t_0) \implies (v_0 = v_1))$ is stored. Similarly, $(5 \neq \mu(t_1))$, so the instance $((5 = t_1) \implies (v_0 = v2))$ is stored. $(\mu(t_0) = \mu(t_1)) \wedge (\mu(v_1) \neq \mu(v_2))$, so the $\mathcal{FCC}$ instance $((t_0 = t_1) \implies (v_1 = v_2))$ is asserted.

The SAT solver is called, and the process of checking the model and asserting extra $\mathcal{FCC}$ instances iterates. ■

The number of clauses that are asserted for an $\mathcal{FCC}$ instance depends on whether any values are known. If an index is a constant, then fewer clauses are asserted by variants of Algorithm 5.2.

Example 5.9

The $\mathcal{FCC}$ instance $(2 = i^{[n]}) \implies (w^{[m]} = 0)$ can be encoded as $(\neg i[n-1] \wedge \ldots \wedge i[1] \wedge \neg i[0]) \implies e$, and $\forall_{k=0}^{m-1}(e \implies \neg w_k)$. This encoding has $m + 1$ clauses and 1 fresh variable, versus the $1+2n+2m$ clauses and $n+1$ fresh variables in all variables case. ■

If the result at each possible index is known, and there are other *select* expressions. Then it is not necessary to create $\mathcal{FCC}$ instances for every distinct pair of *selects*. The next example shows an instance this occurring. Of the approaches we discuss, the $\mathcal{DCI}$ approach is the best at avoiding generation of unnecessary $\mathcal{FCC}$ instances.

Example 5.10

Consider a problem where: $\forall_{0 \leq i < 16}(select(a^{[4:5]}, i) = i)$, $select(a^{[4:5]}, t_0^{[4]}) = t_1^{[5]}$, and $select(a^{[4:5]}, t_2^{[4]}) = t_3^{[5]}$. The constant indices specify the array's value entirely, so there is no need to enforce $\mathcal{FCC}$ instances between the *selects* at t_0 and t_2; it is enough to assert them between each of the constant indices and t_0 and t_2. ■

The $\mathcal{A}bsref$ implementation does not detect and share clauses between $\mathcal{F}CC$ instances that are completely or partially identical. Of our implementations, only $\mathcal{A}ck_{ite}$ will detect and share clauses between $\mathcal{F}CC$ instances.

Example 5.11

If $(select(a_0, t_0) = select(a_0, t_1))$ and $(select(a_1, t_0) = select(a_1, t_1))$, then two $\mathcal{F}CC$ instances have the same left side. Assume $select(a_0, t_0)$ is replaced by v_0, $select(a_0, t_1)$ by v_1, $select(a_1, t_0)$ by v_2, and $select(a_1, t_1)$ by v_3. Then the $\mathcal{F}CC$ instances are $((t_0 = t_1) \implies (v_0 = v_1))$, and $((t_0 = t_1) \implies (v_2 = v_3))$. The CNF conversion algorithm (Algorithm 5.2) will create duplicate fresh variables to represent that the indices are bitwise equal. ■

$\mathcal{A}bsref$ as implemented by STP2 avoids adding all the $\mathcal{F}CC$ instances, but brings extra problems. The SAT solver may arbitrarily set variables that violate the $\mathcal{F}CC$ when they could have easily been set so that the $\mathcal{F}CC$ holds, requiring unnecessary refinement iterations.

Example 5.12

Consider two *selects*, both of which have indices evaluating to 6, with the corresponding results respectively:$\langle 110 \rangle$ and $\langle 11\star \rangle$. Because the indices are the same, the values must be the same. So the SAT solver must choose a 0 for the final $\star$ value. However, because some $\mathcal{F}CC$ instances have been omitted, it may be set to 1, requiring an extra refinement iteration. ■

In STP2's $\mathcal{A}bsref$ implementation, after finding a candidate SAT model, all the assignments to SAT variables are discarded. This is the standard behaviour of Minisat's programmatic interface. For instance, if there are 1 million clauses, after the refinement phase asserts extra clauses, the SAT solver finds a satisfying assignment to these 1 million clauses. On a single core of an Intel Q8400 Linux computer, Minisat 2.2 takes about 150ms to do so. So on such problems, STP 0.1 and STP2 is limited to about 7 refinement iterations per second.

Other than the cost of checking the $\mathcal{FCC}$ and making extra SAT solver attempts, $\mathcal{A}bsref$ risks that the over-approximated problem might be more expensive to solve than the original problem.

Consider a hard bit-vector problem conjoined with an easily unsatisfiable array problem, where the array part is: $(select(a, t_0) \neq select(a, t_1))$ where t_0 and t_1, in some opaque manner, are equivalent but syntactically different. Because abstraction replaces each *select* expression by fresh variables v_0 and v_1, the array part becomes $v_0 \neq v_1$, which is trivially satisfiable. So the SAT solver must solve the *hard* bit-vector problem before the problem can be refined. Because the array part of the problem has been abstracted away, and will not be refined until a satisfiable model is produced to the bit-vector problem, the easily unsatisfiable array part is essentially ignored until after a satisfying assignment is found.

We could reduce the overhead per refinement iteration, by allowing clauses to be asserted to the SAT solver during search, as we do in the $\mathcal{DCI}$ approach. However, preventing the SAT solver from being forced to solve a more difficult abstracted problem requires the SAT solver to have information about the $\mathcal{FCC}$. We discuss an approach that does this next.

5.5 Eliminating *selects*: Delayed Congruence Instantiation

The Delayed Congruence Instantiation ($\mathcal{DCI}$) approach, as we use the term, operates inside the SAT solver. $\mathcal{DCI}$ asserts $\mathcal{FCC}$ instances incrementally, as the indices progressively are assigned the same value. When asserting $\mathcal{FCC}$ instances, $\mathcal{DCI}$ preserves the SAT solver's partial assignment which our $\mathcal{A}bsref$ implementation discards. Similarly to the $\mathcal{A}ck$ approach, the $\mathcal{FCC}$ is always enforced—but without the requirement to add every $\mathcal{FCC}$ instance upfront. $\mathcal{DCI}$ thus attempts to overcome the problems of $\mathcal{A}ck$, which introduces $\mathcal{FCC}$ instances too early, and $\mathcal{A}bsref$, which introduces $\mathcal{FCC}$ instances too late. A disadvantage, however, is that the implementation of $\mathcal{DCI}$ is intimately tied with a particular SAT solver's implementation.

One way to think of this approach is that the SAT solver operates on an implicit CNF. $\mathcal{FCC}$ instances are instantiated only when they are likely to contribute to unit propagation. So instead of the SAT solver using memory to store clauses which do

Algorithm 5.5 Precursor phase to applying $\mathcal{DCI}$

```
Require: $e$, a formula
1: Create selects, a set of tuples of an array literal, an index, and a result
2: $selects \leftarrow \{\}$
3: for all $select(a, i) \in subterms(e)$ do                    // in topological order
4:     Replace $select(a, i)$ in $e$ by a fresh variable $v$
5:     Add $\langle a, i, v\rangle$ to the selects
6: end for
7: Assert $e$ to the SAT solver
```

not participate in unit propagation, the clauses are stored compactly as lists of *select* indices and results.

Another way to think of the approach is as a form of $\mathcal{A}bsref$ which is able to enforce that the $\mathcal{FCC}$ is satisfied on partial assignments. The initial CNF asserted to the SAT solver, and the final CNF, in the worst case are identical for $\mathcal{DCI}$ and $\mathcal{A}bsref$.

The precursor steps to applying $\mathcal{DCI}$ are given in Algorithm 5.5. First, fresh variables replace *select* terms. Second, after replacing all the *select* terms with fresh variables, the problem is asserted to the SAT solver. To enforce the $\mathcal{FCC}$, the $\mathcal{DCI}$ algorithm must know which of the SAT solver's variables, if any, correspond to particular indices and results. So, when the $\mathcal{DCI}$ algorithm is invoked, it is told which variables in the CNF correspond to the indices and results of *select* terms.

Let us briefly review SAT solver terminology which was introduced in chapter 2. A SAT solver performs *unit propagation*, which assigns variables that are entailed by the current assignment. SAT solvers also perform *search*, which heuristically selects an unassigned variable, and assigns it a truth value. Search is performed only when unit propagation is at a fixed point. When assignments are made, the *decision level* is the number of assignments set via search. A *conflict* is when the assignment is inconsistent. A *cancel* undoes the work performed beyond a decision level. A *trail* is a list of pairs (v, ℓ), where v is a variable that has been assigned a value, and ℓ is the decision level at which it happened.

The $\mathcal{DCI}$ algorithm (Algorithm 5.6) we first present is simplified to highlight its key features. A more efficient and complete implementation is given later (Algorithm 5.10). $\mathcal{DCI}$ is run alternately with unit propagation, until neither causes any changes. Then search is performed. The $\mathcal{DCI}$ algorithm does not change the assignments to variables, it simply asserts $\mathcal{FCC}$ instances. If the indices of two

Algorithm 5.6 A simple $\mathcal{DCI}$ algorithm. Run after unit propagation inside the SAT solver.

```
Require: selects          // A set of tuples of an array literal, an index, and a result
Require: knownIndices  // A map from array literals to a map from integer indices
    to a list of ⟨index, result⟩ pairs.
 1: procedure DCI(selects, knownIndices)
 2:     for all ⟨array, k, index, result⟩ ∈ knownIndices do
 3:         if $\mu(index)$ contains a $\star$ value then
 4:             if knownIndices[array][k][0] = ⟨index, result⟩ then
 5:                         // Instantiate the $\mathcal{FCC}$ between the new 0th and others
 6:                 for all $i \in 2 \ldots (size(knownIndices[array][k]) - 1)$ do
 7:                     FCC_INSTANCE(knownIndices[array][k][1], knownIndices[array][k][i])
 8:                 end for
 9:             end if
10:             remove ⟨index, result⟩ from knownIndices[array][k]
11:         end if
12:     end for
13:     for all ⟨array, index, result⟩ ∈ selects do
14:         if $\mu(index)$ contains no $\star$ value then
15:             Create: integer $k \leftarrow \mu(index)$
16:             if ⟨index, result⟩ ∉ knownIndices[array][k] then
17:                 knownIndices[array][k].add(⟨array, index, result⟩)
18:                 if size(knownIndices[array][k]) > 1 then
19:                             // $\mathcal{FCC}$ instance between 0th and newly assigned
20:                     FCC_INSTANCE(knownIndices[array][k][0], ⟨index, result⟩)
21:                 end if
22:             end if
23:         end if
24:     end for
25: end procedure
```

selects have the same assignment, but the results have different assignments, then the $\mathcal{DCI}$ algorithm will generate a conflict. After asserting an $\mathcal{FCC}$ instance, unit propagation is applied, the process repeats until neither cause a change. Then search is performed.

$\mathcal{DCI}$ introduces $\mathcal{FCC}$ instances between indices when they first have the same propositional assignment. For each array, it maintains a list of *selects* with completely specified indices, that is, the index contains no $\star$ values. When the final bit of an index is assigned, a lookup is performed to find other indices with the same assignment. If the index is currently the only index with that assignment, no $\mathcal{FCC}$ instance is asserted. However, if other indices evaluate to the same integer with the assignment, then an $\mathcal{FCC}$ instance is asserted.

The *knownIndices* map of Algorithm 5.6 holds a list of pairs which have indices that are currently assigned to the same integer. If l, where $l > 1$ pairs have the same index assignment, then the algorithm instantiates $l - 1$ $\mathcal{FCC}$ instances, between the zeroeth pair and each other pair.

Each index that is completely assigned, is stored in the *knownIndices* map. For each integer value, a list of the index/result pairs where the index evaluates to that same integer value is stored. Algorithm 5.6 maintains the invariant that for every k there is an $\mathcal{FCC}$ instance asserted between $knownIndices[k][0]$ and $knownIndices[k][i]$, such that $i > 0$.

An unapparent property of Algorithm 5.6 is the following: Consider the indices stored in the list $knownIndices[array][k]$. The decision levels at which they became totally assigned is monotonically increasing. Hence, when removing indices from the *knownIndices[array][k]* list, it is not necessary for the procedure to consider the impossible case when the zeroeth element of a list is removed, and other elements remain that require $\mathcal{FCC}$ instances to be asserted between them.

Example 5.13

Consider three pairs i_0, i_1, i_2 stored at $knownIndices[array][k]$. Two $\mathcal{FCC}$ instances are asserted between these pairs when they are added to the list, that is, between i_0 and i_1, and between i_0 and i_2. If i_0 could be removed from the list, while i_1 and i_2 remained, then it would be necessary, when i_0 was removed, to assert an $\mathcal{FCC}$ instance between i_1 and i_2. However, because the decision level at which the indices were fully assigned is monotonically increasing, that is, i_0 has the lowest, or the equal lowest decision level, it is not possible for i_0 to be removed, while the others remain in the list. ■

The more straightforward approach is to instantiate the $\mathcal{FCC}$ between all the *selects* when an index first become known. By asserting the $\mathcal{FCC}$ instances just between the zeroeth entry and the others, we hope to sometimes avoid asserting quadratically many $\mathcal{FCC}$ instances.

Example 5.14

Suppose the array $a^{[2:3]}$ appears in 3 *selects* $\{\langle i_0, v_0\rangle, \langle i_1, v_1\rangle, \langle i_2, v_2\rangle\}$. Here the *i*s are vectors of SAT variables corresponding to the *selects'* indices, and the *v*s are the SAT variables corresponding to the fresh variables that replaced the *selects*.

Suppose the assignment to the indices is $\mu(i_0) = \langle 1\star\rangle, \mu(i_1) = \langle 10\rangle, \mu(i_2) = \langle 1\star\rangle$, and assume that the SAT solver replaces the $\star$ assignment of i_0 by 0. As the assignments to the indices i_1 and i_0 are now equal, an $\mathcal{FCC}$ instance is asserted, namely $(i_1 = i_0) \implies (v_1 = v_0)$. We cannot assert that $(v_1 = v_0)$ because the assignments to i_1 and i_0 might be changed later.

If the $\star$ assignment of i_2 is replaced by 0, then another $\mathcal{FCC}$ instance is asserted: $(i_1 = i_2) \implies (v_1 = v_2)$. Note that in this case, 3 indices have the same assignment, and only 2 $\mathcal{FCC}$ instances have been asserted. ■

We should say that our implementation of the $\mathcal{DCI}$ approach sometimes asserts $\mathcal{FCC}$ instances *after* they would first have been useful. This is because $\mathcal{FCC}$ instances are added only after an index is entirely known. As a result, the SAT solver may assign literals wrongly, even when bits are deducible from the values assigned to other indices and values. This causes the SAT solver to perform extra work. The next example clarifies this point.

Example 5.15

Suppose the array $a^{[3:2]}$ appears in three *selects*, $\{\langle i_0, v_0\rangle, \langle i_1, v_1\rangle, \langle i_2, v_2\rangle\}$, and suppose the assignments are $i_0 = \langle 10\star\rangle, v_0 = \langle 0\star\rangle$, $i_1 = \langle 100\rangle, v_1 = \langle 00\rangle$, $i_2 = \langle 101\rangle$, and $v_2 = \langle 00\rangle$. Then the $\star$ value of v_0 must be 0, because if i_0 is $\langle 100\rangle$, it is forced to be 0, and if i_0 is $\langle 101\rangle$ it is also forced to be 0. ■

Our $\mathcal{DCI}$ implementation may also assert some $\mathcal{FCC}$ instances *before* they are useful. As the next example shows, $\mathcal{FCC}$ instances can be asserted even if the values assigned to the results of two *selects* are identical.

Example 5.16

Suppose the array $a^{[2:2]}$ appears in two *selects* $\{\langle i_0, v_0\rangle, \langle i_1, v_1\rangle\}$. If both results are completely assigned, with $\mu(v_0) = \mu(v_1)$, and $\mu(i_0)$ becomes equal to $\mu(i_1)$, then the $\mathcal{F}CC$ instance $((i_0 = i_1) \implies (v_0 = v_1))$ is asserted, even though it does not yet enable unit propagation. ■

The $\mathcal{D}CI$ algorithm that we have presented so far is inefficient. We now describe a version of the $\mathcal{D}CI$ algorithm which more closely matches our $\mathcal{D}CI$ implementation (Algorithm 5.8 – Algorithm 5.10).

We give the $\mathcal{D}CI$ algorithm in three parts corresponding to procedures that we built into the SAT solver in three places. The precursor procedure (Algorithm 5.8) initialises the state that the other procedures use; it is run before SAT solving begins. The second runs after the SAT solver's cancel function, which deletes assignments to variables (Algorithm 5.9). A third procedure runs after unit propagation (Algorithm 5.10).

Algorithm 5.7 shows where the $\mathcal{D}CI$ procedures fits in the SAT solver.

An improvement to the simple $\mathcal{D}CI$ algorithm we presented (Algorithm 5.6) is to use a one-watched literal scheme to determine when the final variable of an index is assigned. This corresponds to line 7 in Algorithm 5.10. After unit propagation, the list of variables that were recently assigned is iterated through. The *trail* is recorded by unit propagation; it records which variables have been set. By iterating through the trail, it is easy to check if a watched variable has been assigned. If the watched literal has been assigned, then each of the variables of the index are checked in turn to see if they are unassigned. If some other index variable is unassigned, then the watchlist is updated to refer to that variable. However, if no unassigned index variables remain, then the index is entirely assigned, so an $\mathcal{F}CC$ instance is asserted.

Another improvement is to record if $\mathcal{F}CC$ instances have been asserted between a pair of *selects* already. Before outputting $\mathcal{F}CC$ instances, a check is performed so that duplicate $\mathcal{F}CC$ instances are not asserted.

Algorithm 5.7 The $\mathcal{DCI}$ algorithm implemented with a SAT solver. Given for the one array case.

```
Require: $e$ a formula
 1: Create integer $decision\_level \leftarrow 0$          // the integer decision level
 2: Create $\mu$                                // assignments from variables to: $\{0, 1\}$
 3: Create $selects$, a set of index/result pairs
 4: $selects \leftarrow \{\}$
 5: for all $select(a, i) \in subterms(e)$ do            // in topological order
 6:     Replace $select(a, i)$ in $e$ by a fresh variable $v$
 7:     Add $\langle i, v\rangle$ to the $selects$
 8: end for
 9: Convert $e$ to CNF and assert to the SAT Solver
10: PERFORM_PRECURSOR($selects$)                           // Algorithm 5.8
11: while some variable is not in $\mu$ do
12:     while the size of the trail changes, and no conflict do
13:         Perform unit propagation
14:         ASSERT_FCC( )                                 // Algorithm 5.10
15:     end while
16:     if a conflict occurred then
17:         Analyse the conflict
18:         Assert the conflict clause
19:         if $decision\_level = 0$ then
20:             return unsatisfiable
21:         end if
22:         Undo assignments until $\mu$ is not in conflict
23:         Update the $decision\_level$
24:         DELETE_WATCHED( )                              // Algorithm 5.9
25:     else
26:         if some variable is not in $\mu$ then
27:             Set a variable not in $\mu$ to 1 or 0
28:             Increment the $decision\_level$
29:         end if
30:     end if
31: end while
32: return satisfiable
```

Example 5.17

Suppose the assignment to some index is $\langle 111 \star \star \rangle$, and that the zeroeth literal is being watched. If, after unit propagation, the assignment becomes $\langle 111 \star 1 \rangle$, then, the watched literal is no longer unassigned, so an iteration through each of the variables is performed. In this case, because the first literal is $\star$, it will become the new watched literal. ■

Algorithm 5.8 Precursor steps for an improved $\mathcal{DCI}$ algorithm. Given for the one array case.

```
Require: pairs, a lists of pairs of bit-vector indices and results
 1: procedure PERFORM_PRECURSOR(pairs)
 2:     Create integer $checked \leftarrow 0$
 3:     Create dci_watchlist, a map from index variables to index/result pairs
 4:     for all $\langle index, result\rangle \in pairs$ do
 5:         $dci\_watchlist[v].add(\langle index, result\rangle)$, where $v \in index \wedge \mu(v) = \star$
 6:     end for
 7:     Create dci_trail, a list of tuples of decision level, integer assignment, index,
        result variables
 8:     Create knownIndices, a map from integer indices to lists of index/result pairs
 9:     Create asserted, a set of $\mathcal{FCC}$ instances that have already been asserted
10:     $knownIndices \leftarrow asserted \leftarrow \{\}$
11: end procedure
```

Algorithm 5.9 Steps performed after cancel for an improved $\mathcal{DCI}$ algorithm. Uses the variables defined in Algorithm 5.8. Given for the one array case.

```
 1: procedure DELETE_WATCHED                    // This runs after the cancel function
 2:     while $size(dci\_trail) > 1$ do
 3:         $\langle level, k, index, result\rangle \leftarrow dci\_trail[size(dci\_trail) - 1]$
 4:         if $level < decision\_level$ then
 5:             return
 6:         end if
 7:                                   // At least one index variable is now unassigned
 8:         Delete the last element of the dci_trail list
 9:         Add $\langle index, result\rangle$ to the dci_watchlist
10:         Remove $\langle index, result\rangle$ from $knownIndices[k]$
11:     end while
12: end procedure
```

To speed up removal from the *knownIndices* map, which needs to occur whenever a variable in an index becomes unassigned, the decision level at which an index became entirely assigned is stored. In the algorithms we present, this is the role of the *dci_trail*. The algorithm that runs during cancellation (backtracking) Algorithm 5.9, uses the *dci_trail* to efficiently remove entries from the *knownIndices* map.

An invariant of this improved version is that the decision levels at which the indices stored in *knownIndices*[*k*] became fully assigned is monotonically increasing. Because DELETE_WATCHED is performed from higher to lower decision levels, the *knownIndices*[*k*][0] element will only be removed from the list when it is the only element in the list. Owing to this invariant, it is not necessary for the DELETE_WATCHED

Algorithm 5.10 Steps after unit propagation for an improved $\mathcal{DCI}$ algorithm. Uses the variables defined in Algorithm 5.8. Given for the one array case.

```
1:  procedure ASSERT_FCC                                  // This runs after unit propagation
2:      for $k = checked \dots (size(trail) - 1)$ do
3:          if $dci\_watchlist(trail[k])$ then
4:              $index \leftarrow dci\_watchlist(trail[k]).index$             // the index variables
5:              $result \leftarrow dci\_watchlist(trail[k]).result$           // the result variables
6:              Delete $trail[k]$ from the $dci\_watchlist$
7:              if another index variable $v_j \in index$ is $\mu(v_j) = \star$ then
8:                                                            // Another variable is unassigned
9:                  $dci\_watchlist.add(v_j, index, result)$
10:             else
11:                                                           // No unassigned index variables
12:                 Create integer $k \leftarrow \mu(index)$
13:                 Add $\langle index, result\rangle$ to $knownIndices[k]$
14:                 $dci\_trail.add(\langle decision\_level, k, index, result\rangle)$
15:                 if $(size(knownIndices[k]) > 1) \wedge ((knownIndices[k][0], select) \notin asserted)$ then
16:                                                        // Have not asserted the FCC already
17:                     $asserted.add(knownIndices[k][0], select)$
18:                     FCC_INSTANCE$(knownIndices[k][0], select)$
19:                     if the SAT solver is now in conflict then
20:                                          // The indices are the same, but the results different
21:                         return a conflict clause
22:                     end if
23:                 end if
24:             end if
25:         end if
26:     end for
27: end procedure
28: $checked \leftarrow size(trail)$                          // Track how much of the trail is checked
```

procedure to consider the case when the zeroeth element of a list *knownIndices*[*k*] is removed. The list will always be empty in that case.

When assignments are cancelled, entries are removed from the *knownIndices* map until the decision level of the assignment equals the current decision level. When the decision level is less than the level at which the index became entirely assigned, then that index is no longer entirely assigned, so the entry must be removed from the *knownIndices* map.

5.6 Evaluation

We base our evaluations on the SMT-LIB `QF_ABV` problems[1] as at the 1st of September 2011. We removed the *BrummayerBiere3* family, because they are crafted problems that test an unconstrained array simplification which STP2 does not implement. Next, we removed problems containing array extensionality. Finally, we removed 22 problems that use index bit-widths greater than 64-bits, which is currently a limit of our $\mathcal{DCI}$ implementation. Our $\mathcal{DCI}$ implementation uses native machine integers rather than arbitrary precision integers to store the known indices (Algorithm 5.10). We were left with 9796 problems.

Compared to SMT-COMP 2011, first, we do not use a problem scrambler. The problem scrambler randomly applies simple transformations like swapping the arguments to commutative operations to make cheating via pattern matching harder in the competition. The problems are unscrambled because scrambling produces results that are harder to reproduce. Second, we are using problems with an unknown satisfiability status. Third, we have excluded problems with array extensionality. Finally, we have not checked whether the satisfiability of problems depends on the semantics of division by zero. We compared the solvers' results for each problem. When multiple solvers answered the same problem they always had the same answer.

In general, the problems have few *selects* with indices that may be equal. The largest number of *selects* from a single array is 6096, but all those indices are constants, meaning that no $\mathcal{FCC}$ instances are generated. Of problems that could be successfully converted to CNF, the most $\mathcal{FCC}$ instances added by $\mathcal{A}ck_{ite}$ was 9600. Problems in the *check* family required more $\mathcal{FCC}$ instances, but they both exceeded the memory limit on all solvers.

5.6.1 A Comparison of Two $\mathcal{A}ck$ Implementations

We compare the performance of $\mathcal{A}ck_{ite}$ and $\mathcal{A}ck_{cnf}$ combined with two SAT solvers. An advantage of $\mathcal{A}ck$ is that any SAT solver that reads DIMACS CNF format can easily be used. Minisat 2.2, the default SAT solver of STP2, placed sixteenth of the twenty-six solvers at the 2011 SAT Competition in the Application UNSAT+SAT

[1]These are contained in the `QF_AUFBV` category, which contains no problems with Uninterpreted Functions (UF).

		$\mathcal{A}ck_{cnf}$		$\mathcal{A}ck_{cnf}+\mathcal{G}$		$\mathcal{A}ck_{ite}$		$\mathcal{A}ck_{ite}+\mathcal{G}$	
Family	#	time	fail	time	fail	time	fail	time	fail
bench_ab	119	**0**		2		1		4	
brummayerbiere	56	912	27	**766**	**25**	1177	26	864	25
brummayerbiere2	22	931	1	783		566	1	**480**	
calc2	16	639		298		614		**287**	
check	2	**0**	**2**	**0**	**2**	**0**	**2**	**0**	**2**
dwp_formula	1750	373	1	**659**		161	1	713	
egt	7719	**37**		95		53		112	
platania	20	217		**208**		230		714	
stp	40	**335**	**1**	838	1	359	1	791	1
stp_sample	52	**2**		3		**2**		3	
Sum	9796	3447	32	3652	28	3163	31	3969	28
Time w/ penalty		19479s		17680s		18694s		17997s	

Table 5.1: $\mathcal{G}$ is Glucose. For each family and solver: 'time' is the time in seconds that solved problems took to complete; 'fail' is the number of problems that exceeded the time limit or memory limit. The 'Time w/ penalty' is the sum of 'time' plus 501 seconds per failed problem. '#' is the number of problems in a family.

division. We also evaluate with the first placed Glucose 2.0 [AS09] SAT solver . We use the same Glucose configuration as was submitted to the SAT Competition; it calls the SatELite [EB05] CNF simplifier before solving.

Tests were run using STP r1656 with a memory limit of 3GB and a time limit of 500 seconds on a single core of an Intel E5507 Linux computer.

STP2's default strategy, which we disabled, is to perform $\mathcal{A}ck_{ite}$ upfront for problems with few array expressions.

Table 5.1 shows the results for the four $\mathcal{A}ck$ configurations. The combinations with SatELite and the Glucose SAT solver answer 3 or 4 more problems than does Minisat 2.2. This demonstrates the advantage of being able to easily upgrade SAT solver. Otherwise there are only small differences in performance.

5.6.2 A Comparison to Other Solvers

Next we ran Boolector 1.5.23, Sonolar r2483, Z3 3.2, and STP2 r1656 with both $\mathcal{A}bsref$ and $\mathcal{D}CI$ on the selected SMT-LIB problems. Boolector won the 2011 SMT-COMP `QF_ABV` division, Z3 was second, and Sonolar was third. We also include the fastest $\mathcal{A}ck$ variant. The results are shown in Table 5.2.

All variants of STP2 solve at least 10 more problems than Boolector 1.5.23, the nearest competitor.

	Boolector		Sonolar		$\mathcal{A}bsref$		$\mathcal{A}ck_{cnf}$+$\mathcal{G}$		$\mathcal{DCI}$		Z3	
Family	time	fail	time	fail	time	fail	time	fail	time	fail	time	fail
bench_ab	6	1	5		**0**		2		**0**		2	
brummayerbiere	996	27	1313	29	830	27	**766**	**25**	639	28	1093	25
brummayerbiere2	799	6	790	7	604	1	**783**		737	1	957	14
calc2	1283		493	4	620		**298**		611		614	4
check	**0**	**2**	**0**	**2**	**0**	**2**	**0**	**2**	**0**	**2**	**0**	**2**
dwp_formulas	677	4	247	4	144		659		**139**		990	8
egt	407		964		32		95		**31**		121	
platania	94		1922	4	187		208		230		**6**	
stp	1072	2	922	1	421	1	838	1	**328**	**1**	1516	15
stp_samples	4		3		**2**		3		**2**		3	
Sum	5338	42	6658	51	2839	31	3652	28	2718	32	5303	68
Time w/ penalty	26380s		32209s		18370s		17680s		18750s		39371s	

Table 5.2: $\mathcal{G}$ is Glucose. For each family and solver: 'time' is the time in seconds that solved problems took to complete; 'fail' is the number of problems that exceeded the time limit or memory limit. The 'Time w/ penalty' is the sum of 'time' plus 501 seconds per failed problem. See Table 5.1 for the number of problems in each family.

Solver	Native	Pre-process & Solve
$\mathcal{A}bsref$	0.5s	0.7s
$\mathcal{DCI}$	0.5s	0.7s
$\mathcal{A}ck_{cnf}$+glucose	3.2s	1.2s
Boolector 1.5.23	33.3s	5.0s
Sonolar 2483	36.5s	1.8s
Z3 3.2	$> 1500s$	5.1s

Table 5.3: Time in seconds to solve the *countbitstableoffbyone0128* benchmark. The first column give times using the specified solver. The second column gives times using STP2 with $\mathcal{A}ck_{ite}$ as a pre-processor, then running the specified solver.

In particular, the STP2 variants are consistently better than other solvers for the *brummayerbiere2* family. Table 5.3 gives the solvers' times, on a single core of an Intel Q8400 Linux computer, for one of the instances in that family, the *countbitstableoff-byone0128* benchmark. STP2 with $\mathcal{A}bsref$ is more than three thousand times faster than Z3. To try Z3 with $\mathcal{A}ck_{ite}$, we used STP2 as a pre-processor, with most simplification disabled, to parse the problem, structurally hash it, perform $\mathcal{A}ck_{ite}$, and write the result out in SMT-LIB2 format. $\mathcal{A}ck_{ite}$ is the only approach of the four that we investigated that reduces to a bit-vector problem—a format which other solvers can input. The times to perform this preprocessing step and to run each solver are given. Including the time for pre-processing, the other solvers are between six and three hundred times faster. For this problem, clearly the $\mathcal{A}ck_{ite}$ approach is superior to the approaches the other solvers currently implement.

On the *platania* family, Z3 was by far the fastest solver, about 15 times faster than Boolector, and 31 times faster than STP2. The *platania* family is the only one for which STP2 is not competitive.

The *ff.stp* problem, from the STP set, exceeded the 3GB memory limit on all solvers. We reran it on a machine with 64GB of memory. STP2 with $\mathcal{DCI}$ and with $\mathcal{A}ck$ solved the problem in about 1000 seconds using 28GB of memory. The size of the CNF produced was about 95 million clauses in both cases. So many bit-vector constraints, rather than the array constraints, make this problem hard to solve.

Surprisingly, the approach we used to enforcing the $\mathcal{FCC}$ makes little difference to the overall result. The SMT-LIB problems we selected required no more than ten thousand $\mathcal{FCC}$ instances, which was not enough to contrasts the differences between $\mathcal{FCC}$ instantiation approaches. In the next section we evaluate the $\mathcal{FCC}$ approaches on a more difficult array problem.

5.6.3 A Problem Requiring Many $\mathcal{FCC}$ Instances

It is easy to build a more difficult array problem than those we have encountered so far. Given a bit-width n, for 2^n fresh variables $(v_0 \ldots v_{2^n-1})$ we now assert: $\forall_{i \in 0\ldots(2^n-1)}(select(a^{[n:n]}, i) = i \wedge select(a^{[n:n]}, v_i) = i)$.

Example 5.18

With $n = 1$, the constraints are

$$select(a^{[1:1]}, 0^{[1]}) = 0^{[1]}$$
$$select(a^{[1:1]}, 1^{[1]}) = 1^{[1]}$$
$$select(a^{[1:1]}, v_0^{[1]}) = 0^{[1]}$$
$$select(a^{[1:1]}, v_1^{[1]}) = 1^{[1]}$$

■

This problem creates fresh variables and constrains them to equal the value stored at the same index. For instance, v_6 must equal 6. With $n = 8$, there are 512 distinct *select* terms, but 256 of those have constant indices which are obviously not equal to each other. At worst, $(2^{2n} + \frac{2^n(2^n-1)}{2})$ $\mathcal{FCC}$ instances are instantiated by $\mathcal{A}bsref$ and $\mathcal{A}ck$.

Because the result at every constant index is known initially, the $\mathcal{DCI}$ algorithm will only assert $\mathcal{FCC}$ instances between pairs of *selects* with one constant index and one variable index. For $\mathcal{DCI}$, the maximum number of $\mathcal{FCC}$ instances asserted for this problem is 2^{2n}.

For $n = 8$, $\mathcal{A}ck$ and $\mathcal{A}bsref$ will instantiate at most 98,176 instances, and $\mathcal{DCI}$ at most 65,536 instances. We ran the experiments in this section on a single core of a Intel Q8400 Linux computer, with a memory limit of 4GB. We observed these results:

- Z3 3.2: 0.5 seconds, 19MB
- STP2 r1659 with $\mathcal{DCI}$: 0.5 seconds, 32MB, asserting 31,547 $\mathcal{FCC}$ instances
- STP2 r1659 with $\mathcal{A}ck_{cnf}$: 3.4 seconds, 104MB, asserting 98,176 $\mathcal{FCC}$ instances
- STP2 r1659 with $\mathcal{A}bsref$: 17 seconds, 106MB, with 258 refinement iterations, asserting 98,176 $\mathcal{FCC}$ instances
- STP2 r1659 with $\mathcal{A}ck_{ite}$: 19 seconds, 718MB, asserting 98,176 $\mathcal{FCC}$ instances
- Boolector 1.5.23: 41,400 seconds, 109MB, 67,062 refinement iterations

With $\mathcal{A}bsref$, when the refinement limit is reached (line 35 of Algorithm 5.4), about 65,000 $\mathcal{FCC}$ instances are asserted at once to the SAT solver. $\mathcal{A}ck_{ite}$ uses about 7 times more memory than $\mathcal{A}bsref$, even though both versions send the same number of $\mathcal{FCC}$ instances to the SAT solver. This is because $\mathcal{A}ck_{ite}$ performs an expensive AIG to CNF conversion step.

For $n = 10$, $\mathcal{A}ck$ and $\mathcal{A}bsref$ will instantiate at most 1,572,352 $\mathcal{FCC}$ instances, and $\mathcal{DCI}$ at most 1,048,576 instances. We observed these results:

- STP2 r1659 with $\mathcal{DCI}$: 12.1 seconds, 215MB, asserting 510,013 $\mathcal{FCC}$ instances
- Z3 3.2: 22.4 seconds, 56MB
- STP2 r1659 with $\mathcal{A}ck_{cnf}$: 641 seconds, 1929 MB, asserting 1,572,352 $\mathcal{FCC}$ instances

- STP2 r1659 with $\mathcal{A}bsref$: 14,103 seconds, 2780MB, with 2050 iterations, asserting 1,572,352 $\mathcal{F}CC$ instances

- STP2 r1659 with $\mathcal{A}ck_{ite}$: exceeded the 4GB memory limit after 50 seconds.

For $n = 10$, $\mathcal{A}bsref$ asserted 1,047,553 instances all at once after iterating through all *select* indices.

We did not rerun Boolector 1.5.23 at $n = 10$, because it was the slowest on the $n = 8$ instance.

$\mathcal{A}ck_{ite}$ uses about 7 times more memory than $\mathcal{A}ck_{cnf}$ even though the number of $\mathcal{F}CC$ instantiated is the same. This is because $\mathcal{A}ck_{ite}$ bit-blasts to AIGs then undertakes an expensive AIG to CNF conversion phase.

Z3 3.2 uses the least memory of the solvers, and solves problems quickly.

These problems are well suited to $\mathcal{D}CI$ because one side of the $\mathcal{F}CC$ instances is always a constant. In the $n = 10$ case, there only 11 clauses and one fresh variable introduced per $\mathcal{F}CC$ instance.

Because most of the $\mathcal{F}CC$ instances are needed to make this problem satisfiable, the $\mathcal{A}bsref$ approach asserts all of the $\mathcal{F}CC$ instances. At $n = 10$, $\mathcal{A}ck_{ite}$, which asserts the same clausal form as $\mathcal{A}bsref$, was about 20 times faster. Generally $\mathcal{A}bsref$ performs best when most of the $\mathcal{F}CC$ instances are unnecessary.

As the number of clauses that are asserted to the SAT solver grows, $\mathcal{A}bsref$ performs fewer iterations per second. This is because the search is reset each time the SAT solver finds a model. Instead, if the $\mathcal{A}bsref$ implementation asserted clauses to the SAT solver during search (i.e. when the solver is not at decision level 0), like $\mathcal{D}CI$ does, then the number of refinement iterations performed per seconds would increase dramatically.

5.6.4 Quadratic Blow-Up of Select-over-Store Elimination

In this section we compare the performance of `QF_ABV` solvers when solving problems that are specially crafted to quadratically increase in size when select-over-store elimination (section 5.2) is applied. The problems we generate enforce that the index and the result are the same at least at one position of the array. In these problems we arbitrarily fix the index and result's bit-width to 20.

The problems have a chain of *store* expressions, with the same number of *select* expressions reading from them. Note, there is no particular reason for the number of *store* and *selects* to be the same. Given a natural number k, for k fresh variables $(v_0 \ldots v_{k-1})$ assert:

$$S = store(store(\ldots store(a^{[20:20]}, 0, 0) \ldots, k-2, k-2), k-1, k-1)$$

Because there is no array extensionality, here S is a term variable. It allows us to use the same array term in each of the *selects*:

$$\forall_{0 \leq i < k}(select(S, v_i) = v_i)$$

The problem we have defined is contrived to be difficult for solvers which apply Equation 5.4 to eagerly remove *store* expressions.

Example 5.19

For $k = 3$, the problem is:

$$\begin{aligned} S &= store(store(store(a^{[20:20]}, 0, 0), 1, 1), 2, 2) \\ v_0 &= select(S, v_0) \\ v_1 &= select(S, v_1) \\ v_2 &= select(S, v_2) \end{aligned}$$

When select-over-store elimination is applied to the $k = 3$ instance, the problem is transformed into the following constraints:

$$\begin{aligned} v_0 &= ite(v_0 = 2, 2, ite(v_0 = 1, 1, ite(v_0 = 0, 0, select(a, v_0)))) \\ v_1 &= ite(v_1 = 2, 2, ite(v_1 = 1, 1, ite(v_1 = 0, 0, select(a, v_1)))) \\ v_2 &= ite(v_2 = 2, 2, ite(v_2 = 1, 1, ite(v_2 = 0, 0, select(a, v_2)))) \end{aligned}$$

■

Consider an instance where $k = 300$. Initially there are about 600 array expressions. Applying select-over-store elimination increases this to about 90,000 ITE expressions.

To solve with $k = 300$, Sonolar 2217 takes 0.07 seconds, Boolector 1.5.23 takes 1.7 seconds and uses 3.8MB of memory, STP2 r1398 takes 11 seconds and uses 1GB of memory, and Z3 3.1 takes 3900 seconds and uses 740MB of memory.

To solve with $k = 3000$, Sonolar 2217 takes 0.7 seconds, Boolector 1.5.23 takes 202 seconds and uses 36MB, and STP2 r1398 exceeds the 4GB memory limit. We did not re-run Z3 3.1 because it was the slowest at $k = 300$.

As can be seen, there is a large variation in the time and memory used by different solvers. At $k = 300$, Sonolar 2217 is more than 55,000 times faster than Z3 3.1.

The quadratic blow-up due to select-over-store elimination badly affects STP2; it is not able to solve the $k = 3000$ case.

5.6.5 A Comparison with STP 0.1

Ganesh and Dill [GD07] measure STP on 12 benchmarks. All of these benchmarks are contained in STP's public repository. The problems range in size from 8MB to 442MB, in total they are 1.5GB. To quantify STP2's improvement we re-ran the measurements on a single core of a Intel Q8400 Linux computer with a memory limit of 5GB. STP 0.1 is the version of STP initially open-sourced; it is downloadable from STP's web site. In Table 5.4 we compare STP 0.1 with STP2 r1656 using $\mathcal{A}bsref$.

STP 0.1 exceeds the 5GB memory limit on one problem, and is faster than STP2 on two problems. Ignoring the memory-out, STP2 is about 7 times faster overall. Using STP2 with $\mathcal{DCI}$ is 10 seconds faster using $\mathcal{A}bsref$. Using STP2 with $\mathcal{A}ck_{ite}$ is about 10 seconds slower than using $\mathcal{A}bsref$. So again, these problems are insensitive to how the $\mathcal{FCC}$ is enforced. In total, STP2 $\mathcal{A}bsref$ spent about 10 seconds performing SAT solving; the rest of the time is spent parsing and simplifying the problem. So further improvements for these problems will likely come from speeding up the parsing and simplification phases.

problem	STP 0.1	STP2 r1656 $\mathcal{A}bsref$
610dd9dc.T	MO	11s
grep0084	68s	4s
grep0095	84s	4s
grep0106	83s	4s
grep0117	94s	4s
grep0777	236s	25s
testcase15	26s	15s
testcase16	28s	17s
testcase20	26s	42s
thumbnailout-noarg.9872	593s	49s
thumbnailout-spin1-2.11493	1121s	94s
thumbnailout-spin1-concreteget	44s	56s
Total	2403s	325s

Table 5.4: Time in seconds for STP 0.1 and STP2 r1656 to solve the problems given in Ganesh and Dill [GD07]. 'MO' is memory-out.

5.7 Related Work

Because arrays model the behaviour of a computer with memory, and because they are a basic operation in many programming languages, their study has a long history. Ackermann [Ack54] realised that a theory with uninterpreted functions and equality can be reduced to a theory with equality by instantiating the $\mathcal{FCC}$.

5.7.1 STP 0.1

Ganesh and Dill [GD07] describe what they call the "array substitution" optimisation. Assume we are given $select(a, c) = t$, where a is an array literal, c is a constant, and t is a term not containing any array terms. The optimisation substitutes the *select* expression throughout the problem by t. As described in section 5.1, STP2 only performs the replacement if t is a constant, which avoids both the expense of traversing the expression t to find any array expressions, and the need to sometimes bit-blast t when it is needed to assert an $\mathcal{FCC}$ instance. The $\mathcal{A}bsref$ implementation is not able to bit-blast extra expressions, only to generate $\mathcal{FCC}$ instances. Extra work is needed to measure the cost and benefits of both approaches on a range of benchmarks.

Ganesh and Dill [GD07] describe the *select* abstraction-refinement algorithm of STP 0.1 which is the same, except for how the $\mathcal{FCC}$ instances are generated, to Algorithm 5.4. They also describe a type of abstraction-refinement for *stores*, which is currently disabled in STP2. As seen in subsection 5.6.4, and as described by Ganesh and Dill, select-over-store elimination can quadratically increase the number of expressions. In some cases their approach can avoid this blow-up, but it complicates the implementation of the other algorithms. The store-absref algorithm is not given in the paper, but an implementation is contained in the STP 0.1 source code. Algorithm 5.11 gives the algorithm.

Algorithm 5.11 STP 0.1 *store* abstraction-refinement algorithm

```
Require: $e$, a formula
1: $original$ a term variable, $original \leftarrow e$
2: Create $now$, $later$, sets of CNF clauses
3: Create $l$, a list of pairs of $select$ expressions, and variables.
4: for all $select(store(\ldots), i) \in e$ do            // after a reverse topological sort
5:     Replace $select(store(\ldots), i)$ in $e$ by a fresh variable $v$
6:     Add $\langle v, select(store(\ldots), i)\rangle$ to $l$
7: end for
8: Apply $\mathcal{A}bsref$ to $e$, if it is unsatisfiable return unsatisfiable
9: if $\mu(original) = 1$ then
10:     return satisfiable
11: end if
12: for all $(v, e) \in l$ do
13:     if $\mu(v) \neq \mu(e)$ then
14:         Add the CNF for $v = e$ to $now$
15:     else
16:         Add the CNF for $v = e$ to $later$
17:     end if
18: end for
19: Assert $now$ to the SAT solver
20: If the SAT solver reports unsatisfiable, then return unsatisfiable
21: Assert $later$ to the SAT solver
22: If the SAT solver reports unsatisfiable, then return unsatisfiable
```

The STP 0.1 approach to store-absref differs from other *store* abstraction approaches, for example Boolector's (described in the next section), in that store-absref results in at most two extra SAT solver calls. If the abstracted problem can be solved, then we are done, otherwise select-over-store elimination is applied and the problem asserted to the SAT solver.

Example 5.20

Consider:

$$select(store(a, t_0, t_1), j) = select(store(a, t_0, t_1), k)$$

If select-over-store elimination is applied, this becomes:

$$ite(t_0 = j, t_1, select(a, j)) = ite(t_0 = k, t_1, select(a, k))$$

Instead store-absref (Algorithm 5.11) asserts the abstracted formula $v_0 = v_1$ to the SAT solver. The original problems is evaluated with the SAT solver's model. If $\mu(v_0) \neq \mu(select(store(a, t_0, t_1), j))$, then: $v0 = ite(t_0 = j, t_1, select(a, j))$, is asserted to the SAT solver. This asserts Equation 5.1 which was previously omitted from the problem. Likewise, if $\mu(v_1) \neq \mu(select(store(a, t_0, t_1), k))$, then $v_1 = select(store(a, t_0, t_1), k)$ is asserted. ■

STP2 has the capability (inherited from STP 0.1 but currently disabled) to sort *stores* where the indices can never be equal using a rule:

$$\text{Given: } store(store(a, i, j), k, l),$$

if $i \neq k$ and i is less than k in a specified total order, rewrite to: $store(store(a, k, l), i, j)$.

That is, if two *store* indices can never be equal, then order the *stores* according to some total order on the index expressions. This normalises terms, but potentially increases the number of terms.

Example 5.21

Consider the formula $v_0 = select(S, v_1)$, where S is a syntactic variable

$$S = store(store(a, (t_1 + 2), t_2), (t_1 + 4), t_3)$$

If another formula is created, say, $v_1 = select(store(S, t_1, t_4), t_5)$, and if the indices are ordered $t_1 \prec (t_1 + 2) \prec (t_1 + 4)$, then, if sorting of indices is performed, the formula will be sorted so that the t_1 index is the innermost. This requires the creation of three *store* expressions, rather than just one. ■

Because of the potential for blowing up the number of terms, we have disabled this feature in STP2.

5.7.2 Boolector

Brummayer and Biere [BB09] describe Boolector's abstraction-refinement algorithm for extensional arrays. Abstraction-refinement as implemented by Boolector [Bru09] is a more sophisticated implementation than STP2's $\mathcal{A}bsref$. In particular, their approach has three features that our $\mathcal{A}bsref$ lacks: it handles array extensionality, it does not perform upfront select-over-store elimination or remove array-ITEs, and it encodes array indices to CNF when they are first needed (like STP 0.1 does).

Earlier versions of Boolector implemented an unconstrained simplification for arrays. That is, if there is only a single occurrence of the array variable a, then $select(a, t)$ is replaced by a fresh variable. In general STP2 does not implement unconstrained variable simplification for arrays but for problems with few array terms, STP2 applies $\mathcal{A}ck_{ite}$ to convert array problems to bit-vector problems early on. This has the same effect as performing the unconstrained array simplification. The *brummayerbiere3* family, which we omitted from our experiments, are crafted benchmarks that are easy if unconstrained elimination of arrays is implemented.

Rather than remove array-ITEs and *stores* upfront, like STP2 does, which may quadratically blow-up, Boolector performs abstraction-refinement which asserts that the array theory axiom holds during the refinement phase. Unlike STP2, which asserts just $\mathcal{FCC}$ instances during refinement, Boolector also asserts instances of the array theory axiom (Equation 5.1), and the extensionality axiom (Equation 5.2). Boolector begins by replacing *selects* terms with fresh variables, then an abstraction-refinement loop checks the $\mathcal{FCC}$, the array axiom, and extensionality axiom.

Brummayer and Biere [BB09] Section 11.6, describe how $\mathcal{FCC}$ instances are encoded; STP2 uses the same CNF encoding which we presented as Algorithm 5.2.

Boolector asserts the CNF corresponding to an index expression when it is first required in an axiom instance. For example, if the *select* expression $select(a, t)$ is replaced by a fresh variable v, and t appears nowhere else, then t is initially omitted from the CNF. When t is required for an axiom instance, only then is it encoded to CNF. If t is complex, and is not required for an axiom instance, there will be a considerable saving. To simplify our $\mathcal{A}bsref$ algorithm, STP2 encodes all indices to CNF before beginning refinement. The CNF clauses corresponding to all index expressions are asserted to the SAT solver initially, because this makes the $\mathcal{A}bsref$ algorithm simpler.

Example 5.22

Consider the formula

$$select(store(store(a^{[n:m]}, t_0, t_1), t_2, t_3), t_4) = select(store(a^{[n:m]}, t_5, t_6), t_7)$$

Initially each *select* is replaced with a fresh variable, giving: $v_0 = v_1$. In the following we use μ to give the integer value from the SAT Solver's model. If the model returned by the SAT solver is: $\mu(t_4) = 6, \mu(t_2) = 6, \mu(t_3) = 1$ and $\mu(v_0) = 2$, then Equation 5.1 is not satisfied, so Boolector asserts that: $(t_4 = t_2) \implies (t_3 = v_0)$.

If another model is returned where none of the *store* indices equals a *select* index, that is, $((\mu(t_4) \neq \mu(t_2)) \wedge (\mu(t_4) \neq \mu(t_0)) \wedge (\mu(t_7) \neq \mu(t_5)))$, and the result of the *selects* differs, i.e. $(\mu(v_0) \neq \mu(v_1))$, then Boolector asserts: $((t_4 \neq t_2) \wedge (t_4 \neq t_0) \wedge (t_7 \neq t_5)) \implies (v_0 = v_1)$ ■

5.7.3 BAT

Manolios et al. [MSV06] describe the Bit-level Analysis Tool (BAT), an eager bit-vector and extensional array solver. Their insight is that it is practical to enforce array extensionality if the bit-width of indices is reduced—so that arrays can be compared at every possible array index. How many indices the arrays need to have to preserve satisfiability is calculated.

First, they apply the rewrite rules described in section 5.2. Because their problems may contain extensionality, this gives fewer rather than no *stores* and array-ITEs. Then, they count the number of array accesses for each array and for each array that it is transitively related to. Next, the count is increased to allow arrays to differ at some positions. The indices' bit-width is narrowed so that the cardinality contains at least the necessary number of results. A quadratic number of constraints are asserted, to enforce that the same index maps to the same reduced index.

Example 5.23

If there are only two *selects* from an array a: $select(a, t_0^{[32]})$ and $select(a, t_1^{[32]})$, then the indices are narrowed to 1-bit. Two fresh 1-bit variables are created, $v_0^{[1]}$ and $v_1^{[1]}$, and these constraints are asserted $(t_0 = t_1) \implies (v_0 = v_1)$ and $(v_0 = v_1) \implies (t_0 = t_1)$. ■

Example 5.24

Assuming an expressions contains only the following array expressions:

$$select(a_0^{[6:6]}, t_0), select(a_0^{[6:6]}, t_1), select(a_1^{[6:6]}, t_2), select(a_1^{[6:6]}, t_3), a_0 = a_1$$

Then it is always possible to have $(a \neq b)$, because there are 2^6 possible indices for each array, but constraints on only two of those indices per array.

BAT will calculate that is equisatisfiable to the case that uses an index bit-width of 2 bits. Because the result is 6-bits, the equality $(a = b)$ being 1 implies that all 24 bits (4 locations of 6 bits each) of each array are identical. This is less expensive than asserting that all 2^6 locations are identical. ■

Reducing the bit-width of indices is useful because BAT compares all the values stored in arrays to determine whether two arrays are equal. Reducing the number of possible indices makes this comparison practical.

5.7.4 Other Solvers

Biere and Brummayer [BB08a] describe a lazy solver for all-different constraints over bit-vectors. An all-different constraint enforces that the bit-vectors in a set are all assigned different values. We, like them, have a one-watched literal scheme. In the conclusion of their paper they propose $\mathcal{D}CI$.

Integrating clause propagation with specialised reasoning as we do for $\mathcal{D}CI$ has been done previously in other contexts. Chu Min Li [Li00] describes EqSatz, a SAT solver that handles equalities specially. Equalities (bi-implications) between literals are discovered in the CNF, and propagated by inference rules. One of the inference rule given is: $(l_1 \Leftrightarrow l_2 \Leftrightarrow l_3) \wedge (l_1 \Leftrightarrow l_2 \Leftrightarrow l_4)$ implies $l_3 \Leftrightarrow l_4$. This equivalence reasoning is performed after unit propagation, and before search—like we do. Likewise, the SAT solver Cryptominisat [SNC09] extracts exclusive-ors from its CNF input, and applies Gaussian elimination to remove variables.

We were introduced to the idea of generating CNF clauses inside the SAT solver by the "Lazy Clause Generation" approach of Ohrimenko et al. [OSC09].

Ganesh et al. [GOSL+12] describe a SAT solver programmatic interface which allows the trail to be iterated over, and for clauses to be introduced during search. They call the approach of instantiating clauses as needed "online abstraction-refinement". If such interfaces became more expressive, and common across different SAT solvers, it would reduce the coupling between implementations of $\mathcal{DCI}$ and a particular SAT solver.

Bruttomesso et al. [BCF+06] decide whether to include all the $\mathcal{FCC}$ instances upfront or to instead use interface variables to communicate between theory solvers. They determine which approach is better based on the number of extra equalities introduced.

Nelson and Oppen [NO80] give a congruence closure algorithm for solving problems in the theory of uninterpreted functions with equality. Their congruence closure algorithm is good at reasoning about the effect of nested function calls, for instance, that $f(f(f(a))) = a \wedge f(f(f(f(f(a))))) = a$, implies $f(a) = a$. Nieuwenhuis and Oliveras [NO05] give a congruence closure algorithm that can quickly explain which equalities imply another equality. We expect the nesting of *selects* to be too shallow in software verification problems to justify their approaches. That is, it is rare to have deeply nested *selects* like $select(select(select(a, t_0), t_1), t_2)$

Stump et al. [SBDL01] give a refutation procedure for the extensional theory of arrays based on congruence closure which does not use a SAT solver. Brummayer and Biere [BB08b] compare Stump et al.'s algorithm against Boolector's abstraction-refinement approach, finding their abstraction-refinement implementation to be hundreds of times faster. It would have been interesting to explore whether SAT based approaches like $\mathcal{A}ck$ perform much worse than $\mathcal{A}bsref$ approaches when solving extensional array problems. We leave this for later work.

5.8 Conclusion

We focused on approaches to enforcing the $\mathcal{FCC}$, in effect solving problems in the theory of bit-vectors and uninterpreted functions. We found the SMT-LIB benchmarks we selected to be insensitive to the approach chosen. We believe this is because few $\mathcal{FCC}$ instances are required. No SMT-LIB instance needed more than 10,000 $\mathcal{FCC}$ instances. So, even the result of $\mathcal{A}ck$ was a reasonably sized CNF. Reducing, via $\mathcal{A}ck$, to the theory of bit-vectors has the advantages of being able to use: bit-vector theory simplifications, AIG simplifications, CNF simplifications, and of being able to easily use, by some measures, the fastest available SAT solver (currently Glucose 2.0). Using Glucose gave about 10% fewer failures than using Minisat 2.2.

Solvers implement abstraction-refinement of *selects* to avoid necessarily asserting quadratically many $\mathcal{FCC}$ instances. Solvers implement abstraction-refinement of *stores*, and array-ITEs to reduce the chance of a quadratic blowup in the number of expressions. On the SMT-LIB problems we examined, both approaches are unjustified. However, we demonstrated crafted problems showing the consequences of both blow-ups.

On a crafted benchmark (subsection 5.6.3), more $\mathcal{FCC}$ instances are required, so the differences between $\mathcal{FCC}$ instantiation approaches was apparent. In particular, STP2's $\mathcal{A}bsref$ implementation did not perform well when a high proportion of the $\mathcal{FCC}$ instances were required. Because most of the $\mathcal{FCC}$ instances are necessary, it is 20 times faster just to assert all the $\mathcal{FCC}$ instances upfront, rather than to perform refinement phases that gradually approach the effect of $\mathcal{A}ck$. Boolector performed 250 times more refinement iterations than STP2's $\mathcal{A}bsref$ implementation. STP2, inherited from STP 0.1, has an upper bound on the number of refinement iterations that are performed. $\mathcal{A}bsref$ does not perform more refinement iterations than there are distinct *select* expressions (Algorithm 5.4). Significant amounts of time can be saved by placing an upper limit on the number of refinement iterations that are performed. This demonstrates that asserting few clauses per iteration does not guarantee good performance.

We demonstrated an additional advantage of $\mathcal{A}ck$: the ability to easily change the SAT solver. STP with $\mathcal{A}ck_{cnf}$ and Glucose solved the most SMT-LIB problems. This is a reasonable comparison as we started work on our $\mathcal{DCI}$ implementation before Glucose won the competition. It shows an advantage of being able to easily follow improved SAT solver performance. Our $\mathcal{DCI}$ implementation is tied closely with the SAT solver. At present there is no commonly used "low level" programmatic interface to SAT solvers. Such an interface would make it easier to move between SAT solvers. The $\mathcal{A}ck$ approaches with Glucose 2.0 solved the most problems.

We showed crafted examples where $\mathcal{D}CI$ significantly outperformed $\mathcal{A}ck$ and $\mathcal{A}bsref$. The larger the number of redundant $\mathcal{F}CC$ instances, the better the relative performance of $\mathcal{D}CI$. On the SMT-LIB2 problems, the $\mathcal{A}bsref$ approach solved one more problem than $\mathcal{D}CI$ did. In general, we believe the $\mathcal{D}CI$ approach is superior to $\mathcal{A}bsref$, but at present lack the real-world problems to make a convincing case.

We showed that STP2 is a significant advance on STP 0.1, being about 7 times faster (subsection 5.6.5). Although the other solvers that we compared against allow the theory of extensional arrays, it is a quick syntactic check to identify whether a problem has extensionality. The approaches that we have described could be implemented as a special case for problems without extensionality by those solvers.

STP2's default strategy is to perform $\mathcal{A}ck_{ite}$ upfront for problems with a small number of array expressions. Currently the limit is ten. This conversion was disabled in all of the experiments in this chapter. This conversion occurs before the bulk of the bit-vector simplifications have occurred, converting the array part of the problem into a bit-vector problem. If array expressions remain, then $\mathcal{D}CI$ is performed. STP2's current strategy is not ideal for the SMT-LIB problems, but works well for the software verifications problems that STP2 is commonly used to solve.

6 Symbolic Execution for Automated Test Generation

AUTOMATED software verification and testing increases the confidence that programs behave as intended. Research into the mechanisation of the task began more than 30 ago, but waned as the difficulty of the task became clearer; in particular scalability proved to be elusive. But recent advances in constraint solving technology have rekindled optimism. At the same time, the amount of software that needs to be trusted, in particular binary code, is growing.

In this chapter we describe a tool which we call, for no particular reason, MinkeyRink. MinkeyRink performs automated test generation for binary programs via structural fuzzing of unmodified Linux x86 binaries. It was our experience in building MinkeyRink that motivated our work on the STP2 bit-vector and array solver; we described that work in the first part of this dissertation. As we shall show, MinkeyRink depends greatly on efficient and correct bit-vector and array solving.

MinkeyRink analyses machine code programs, and generates problems which bit-vector and array solvers, like STP2, are ideally suited to solving. In the evaluation (section 6.7) we show that the majority of the time taken by our automatic test generator is spent performing bit-vector and array solving.

```
int32_t x;                 // A 32-bit variable.
input(x);                  // Read a value into x.

int32_t y = 4 * x;
if (x !=3 && y == 12)
  print("fail");
else
  print("OK");
```

Figure 6.1: A C-language program that sometimes fails. There are two paths through the program. One path is taken for 3 possible assignments to x, the other path is taken otherwise.

6.1 Background

Fuzzing is commonly used to discover inputs that cause a program to fail. Fuzzers generate random instances of a language for use as inputs to a program. An *oracle* then checks if the input causes the program to fail. If it does, then the input is reported to a programmer for investigation.

Although useful, fuzzers are limited because they do not consider the internal structure of a program. If only a small proportion of inputs can reach a part of the program, a fuzzer is unlikely to randomly generate one of those inputs. For instance, in Figure 6.1, just three of the 2^{32} possible values for x will cause the program to print "fail". *Structural fuzzers* overcome this limitation by analysing the program, and using its structure to generate inputs. Variations of the same approach go by many names: smart-fuzzers, glass-box fuzzers, white-box fuzzers, concolic testing, directed automated random testing, and dynamic symbolic execution.

We, like others (section 6.10), use symbolic execution (SE) as the basis for a structural fuzzer. *Symbolic execution* builds formulae that precisely describe the output state of a program in terms of its inputs. The terminology of symbolic execution (sometimes called symbolic simulation) varies, so let us fix our use of terms. SE makes use of a concept of the *state* of a computation, where a variable's value has been replaced by an expression that denotes a function of the program's input. We denote the instruction pointer register (which holds the address of the next instruction to execute) by IP. A *path* is a sequence of instructions. For a given input, a path can be constructed by running the program and listing the successive IP values. (We assume, without loss of generality, that a program has a single entry point, $IP_{initial}$, and a single exit point IP_{final}.) But even simple programs can have trillions of paths, so a path-by-path analysis is impractical. When instructions are in loops, or called multiple times in procedures, those instructions will occur multiple times in the path. By a *trace* we mean a path and the input and output of the system calls that occur along that computation path. A (symbolic) *state* is a map from locations to expressions, together with the IP, where a *location* may be a register or a memory address. The expressions that are mapped to are `QF_ABV` expressions which may contain symbolic variables.

Symbolic execution may *fork* at some conditional branch instructions. Additionally, a state is equipped with a summary of control-flow history: a *path constraint* (PC) keeps track of the class of inputs that would have caused the same flow of control. A state is *feasible* if it can occur along some path. It is natural to associate a path constraint with a state, the PC being a constraint over the program input variables. The PC describes the inputs that would take the same path through the program, that is, the inputs for which the associated state is valid. States can be seen to form a *state tree*, with branching occurring whenever a conditional branch instruction depends on symbolic values. In the state tree, a parent's path constraint is equivalent to the disjunction of its children's path constraints.

For instance, consider again the program (Figure 6.1) which fails if the input is not equal to 3 and if the input multiplied by 4 is 12. When this program is symbolically executed, a special *symbolic variable* (as distinct from the program's variables) is associated with the values returned from the input procedure. These symbolic variables, and expressions that contain them, are contained in the symbolic state. The symbolic variable, which we call s, is assigned to x in the symbolic state, later when x is multiplied by 4, the symbolic state of y is updated to contain the expression $4 * s$. With symbolic execution the analysis is simplified to reasoning about just two possible paths—a tremendous speed-up.

Symbolic execution attempts to overcome the *state space explosion*. By maintaining a state that describes many possible inputs, it is able to tractably reason about many inputs simultaneously. In the example we just saw, billions of possible inputs were split across just two states.

When the program of Figure 6.1 is run on some concrete input, at any control transfer instructions, such as the `(x !=3 && y == 12)` test, the path constraint is

```
int32_t x;
input(x);
print(12 / x);
```

Figure 6.2: A C-language program that may divide by zero

updated to track which branch was taken. In this example, assuming the condition is 0, the path constraint is updated, from being empty (that is 1), to $(s \neq 3 \wedge (2 \times s) = 12)$.

If the path constraint is negated, then the constraint solver can calculate if there exist inputs which would cause the program to take a different path. For example, given a path constraint of $(s \neq 3 \wedge (4 \times s) = 12)$ and a state of $(y = 4 * s, x = s)$, the solver may give $(y = 12, x = (2^{30} + 3))$. In our simple example, through the source code, there are just two paths through the program. One path is taken on three inputs $(x = 2^{31} + 2^{30} + 3, x = 2^{31} + 3$, and $x = 2^{30} + 3)$, and the other path on all other inputs. A fuzzer that did not consider the structure would run for a long time before discovering an input to cause a failure. A structural fuzzer, however, can stop after exploring two paths; it has explored all possible paths.

If the multiplication in our example did not overflow, then taking the true branch of the conditional would be impossible. This shows that, in reasoning about such programs, considering overflow, as bit-vector solvers do, is important for ensuring correctness.

The C++ language [ISO12] does not define the semantics of signed overflow, so the semantics of the example program is undefined. That is, a C++ compiler is free to return any value for 4×2^{30}. So analysing the source code alone is not enough to define the semantics of the compiled program. Without knowing how the compiler translates this program to machine code, it is difficult to say with certainty what this program does. Analysing the binary, as we do, gives a more complete picture of what the program does, because less is hidden.

Structural fuzzers generate and run fewer inputs per second than traditional fuzzers. This is due to the considerable overhead involved in analysing the program and generating new inputs.

The oracle needs to be more sophisticated if the paths through the program are not apparent and depend on the data. A program with a division by zero, such as Figure 6.2, is an example. In this example, there is only one path through the

```
uint32_t x; // Unsigned 32-bit integer
input(x);
int count = 0;
for (int i = 0; i < 32; i++)
  if ((x >> i)  & 1)
    count = count + 1;
print(count);
```

Figure 6.3: Pop count, with the path explosion.

program, but the program will behave differently depending on whether or not $x = 0$. Because the division by zero does not execute a separate path to the other inputs, an oracle that tests if a program possibly has a division by zero needs to check the symbolic state at any division to check if the denominator could possibly be zero. In this dissertation we do not consider the very real challenges of building oracles.

Unfortunately, the *path explosion* problem afflicts symbolic execution. For instance, Figure 6.3 counts the number of 1-bits in x. Unfortunately, this has as many paths as inputs, since each input causes a different path to be taken. If we use normal symbolic execution on this program, then when we reach the *print*() procedure, the symbolic state will only describe one input. This is bad because the advantage of symbolic execution has been lost, that is, we are reasoning about each of the inputs separately. This is the *path explosion* problem which we address in this chapter.

We investigate a *state joining* approach to making symbolic execution more practical, and in this chapter we describe the challenges of applying state joining to the analysis of unmodified Linux x86 executables. The results so far are mixed, with good results for some code. We describe an algorithm which, in effect, analyses an equivalent program with only a single path. For example, the state joining algorithm will effectively transform the code in Figure 6.3 into that of Figure 6.4, allowing the code to be quickly analysed.

Although relevant to other uses of symbolic execution, we are interested in state joining in the context of the verification of binary executables. Dealing with the path explosion is the key challenge in making symbolic execution useful for software verification.

By combining the results of symbolic execution along all the program's paths, the behaviour of the program is described. Although the number of paths through

```
uint32_t x; // Unsigned 32-bit integer
input(x);
int count = 0;
for (int i = 0; i < 32; i++)
  count += ((x >> i)  & 1);
print(count);
```

Figure 6.4: Pop count, without the path explosion.

a deterministic program is less than the number of its input states, real programs still have intractably many possible paths. In traditional symbolic execution, states are split at control flow instructions, and not subsequently joined, even where the program's control flow merges. State joining addresses the path explosion problem by coalescing states with the same instruction pointer. In the presence of state joining, the state tree becomes a (directed, acyclic) *state graph*.

MinkeyRink dynamically disassembles a given executable. A natural approach is to use symbolic execution on semi-random input to explore the space of possible run-time states. The instructions executed on each branch are combined together to produce, over time, a disassembled version of the binary.

We use dynamic disassembly to incrementally build a partial Control Flow Graph (CFG). It is partial, because other blocks or transitions might have occurred if the results from system calls were different, or a different length of input was used. We use the partial CFG to determine when states can be joined.

section 6.2 explains why we are interested in analysing binary code, rather than source code programs. section 6.3 gives a more detailed example to clarify our approach. The example shows the advantages of state joining, disregarding the practical difficulties (these are discussed in section 6.9). section 6.4 contains a description of the overall algorithm. Without applying simplifications, the symbolic formulae quickly grow unmanageable. section 6.5 describes how Boolean minimisation can be used to reduce a formula's size. section 6.6 describes the specifics of MinkeyRink. In section 6.7 and section 6.8 we compare runs with and without state joining. Mostly state joining helps, but it does produce more cumbersome constraint expressions, which can slow down the analysis. We discuss additional obstacles in section 6.9 and compare our approach to others that tackle symbolic execution's path explosion problem (section 6.10).

6.2 Why Binary Analysis?

Analysing source code, rather than binaries, is the more traditional approach to bug finding. Each approach has its advantages and disadvantages.

Analysing binaries has the advantage that all the components that are used at runtime irrespective of their source language have been converted into a common language, that is machine code, which in most cases has precisely defined semantics. This avoids some of the challenges of analysing source code: of analysing different languages, of obtaining the source code that is used at runtime, of determining how the compiler translated the source code, i.e., which compiler options where used, and of determining which compilers were used.

Source code has the advantage that the type and signedness information is present. From binaries it is not straightforward to determine, for example, which data elements are pointers and which are integers. Defects that are reported by a binary analysis tool are harder for programmers to fix, requiring them to trace back to the source code that produced the machine code.

Source code analysis is not specific to a given binary representation. A defect that is specific to a particular architecture will not be identified by binary analysis on another architecture, for example, if a defect occurs for only one possible memory layout of a procedure's parameters. A source code analysis can consider multiple semantics for the source code, rather than just a single specific semantics. If source code is distributed and users compile the software themselves, it is good to be able to analyse several of the possible binary forms simultaneously, rather than just a particular form.

However, the source code may not be available to analyse. For example, when using closed source components, compilers, libraries, device drivers, and operating systems, a source code analysis will have procedure calls without the corresponding source code. Binary analysis provides the ability to analyse not just to the interface with library functions, or system calls, but right through the operating system kernel to the interaction with hardware.

Large systems are often built using multiple languages. For example, C code may contain inline assembly code, and the interacting components may be difficult to check. Large programs combine components (some closed source) from diverse

```
uint64_t multiply(uint64_t x0,uint64_t y0) {
    uint64_t x = x0, y = y0, z = 0;
    while(x!=0) {
        if (!even(x)) /* (x&1) */
            z += y;
        x = x >> 1;
        y = y << 1;
    }
    return z;
}
```

Figure 6.5: Using shift-and-add to multiply positive integers x_0 and y_0

organisations, and use multiple high level languages. Binaries give a common representation.

6.3 State Joining: A More Detailed Example

To provide the intuition of state joining, we will describe the analysis of the Multiply example in Figure 6.5. The program uses shift-and-add to multiply two non-negative integers $x^{[64]}$ and $y^{[64]}$, leaving the result in $z^{[64]}$. We show C-style code for clarity—all analysis is performed on unmodified executables.

Figure 6.6 shows a control-flow graph for the function, and Figure 6.7 shows the state graph that is produced during joining. For each distinct $x^{[64]}$ value, a Multiply call takes a different path through the function body; these many paths make symbolic execution slow. For 64-bit integers, there are 2^{64} distinct paths through the function. To see this, consider the branching on the even condition (block B_3), which checks the rightmost bit of the $x^{[64]}$ variable. On each iteration, the bits of $x^{[64]}$ are shifted one to the right, giving a new rightmost bit. For a given $x_0^{[64]}$, the resulting symbolic expression for $z^{[64]}$ is an expression valid for every $y^{[64]}$ value. For example, given ($x_0^{[64]} = 2$), the symbolic expression produced for $z^{[64]}$ is ($z^{[64]} = (y_0 \ll 1)$), with a PC that simplifies to ($x_0^{[64]} = 2$). The resulting expression for $z^{[64]}$ describes the output state in terms of the input state. Substituting the input values into the formula for $z^{[64]}$ gives exactly the same result as if the function were run with the same inputs.

Symbolic execution addresses the state explosion problem by reasoning about all the states that take a particular path. In this case there are 2^{64} paths to symbolically

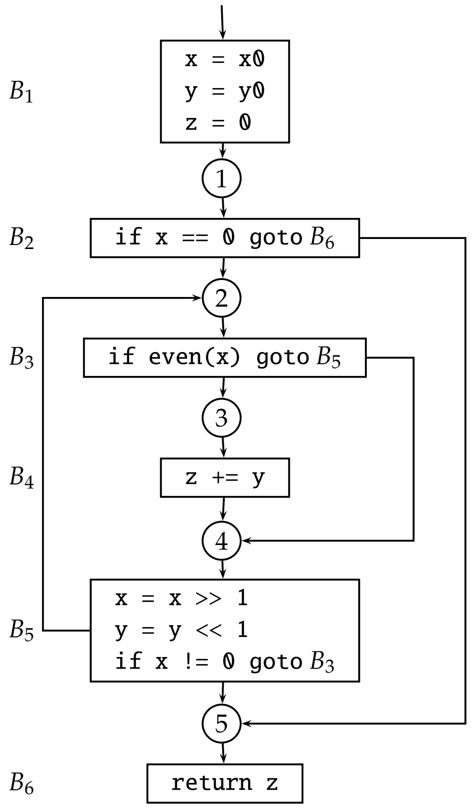


Figure 6.6: CFG for multiplication code

execute, versus $2^{64} \times 2^{64}$ initial states. We now face the path explosion problem—this function still contains 2^{64} paths: too many to reason about one-by-one. With state joining, states with the same IP are joined. This makes good sense for the example, as two paths that have previously differed in whether one bit was one or zero could have identical subsequent paths.

Each time a control transfer instruction is reached, a constraint solver call is made to check which branches can be taken. For example, on reaching block B_2, in Figure 6.6, we call the solver to check which of the branches to points 2 or 5 can be taken. In this example both branches can be taken, so the state is split, with one branch conjoining the PC with $x = 0$, and the other using $x \neq 0$. We may consider the state of the function to have 5 elements: $x^{[64]}$, $y^{[64]}$, $z^{[64]}$, the block that will next be executed (the IP), and the PC. Figure 6.7 shows how the states are split and joined. The topmost state in the figure corresponds to having just entered the function, with the IP at program point 1. The second row results from handling the $x^{[64]} = 0$ condition. One state corresponds to $x^{[64]} = 0$, the other to entering the loop. There are two joins, shown with bold borders. If the states being joined

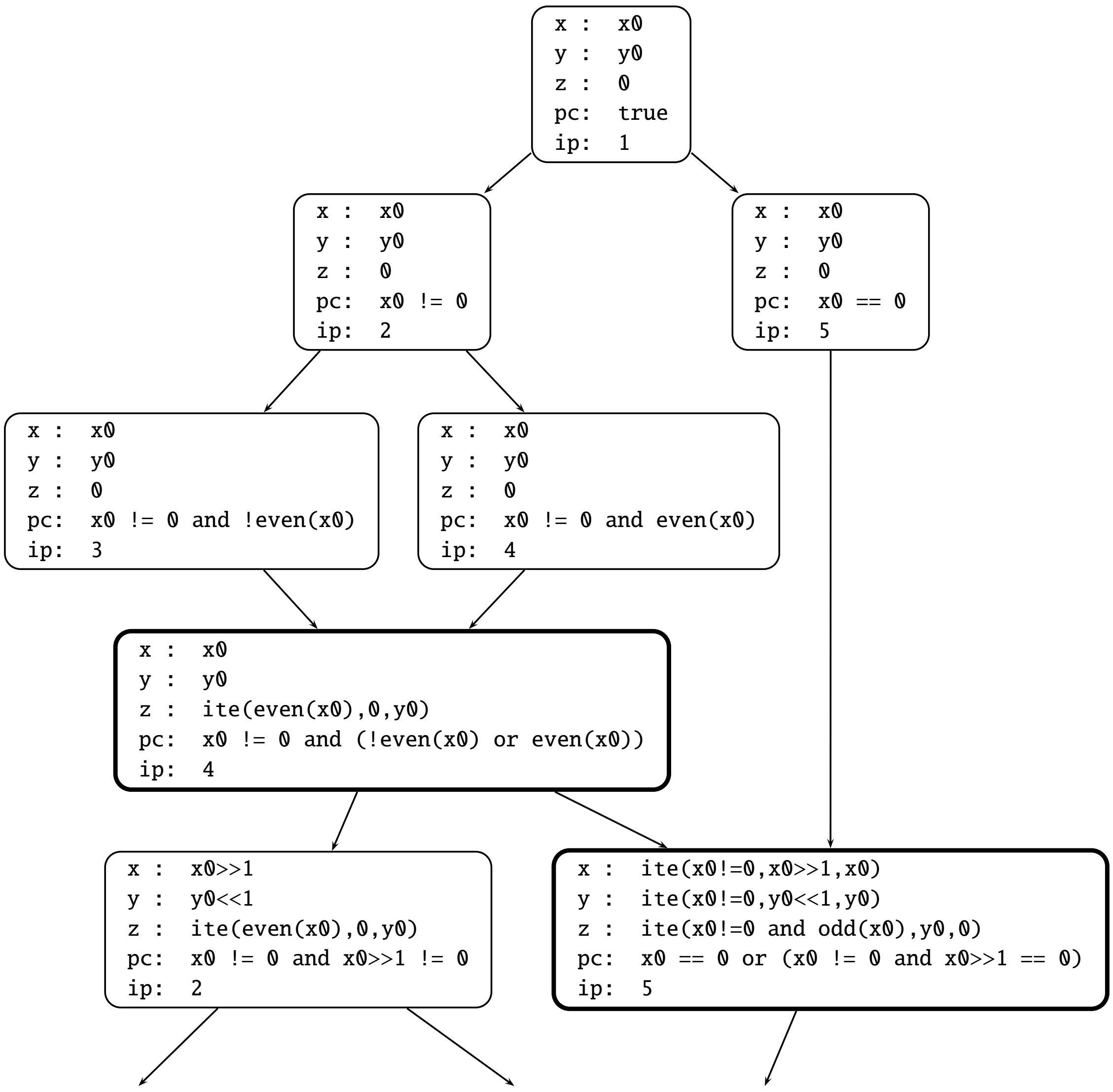


Figure 6.7: The state graph for the example of Figure 6.6

have the same expression, then the value is unchanged, but if they are different, an ITE constructor (for if-then-else) joins the different expressions. The result of a join holds all the information of the component states. No information is discarded.

Eventually $x^{[64]}$'s value at the end of B_5 is $(x_0^{[64]} \gg_l 64)$, so that $x \neq 0$ is unsatisfiable. Then only the branch to point 5 will be taken. When this happens, the final states will be joined together at point 5, producing the complete expression for $z^{[64]}$ in terms of its inputs. The loops are unrolled automatically—no domain knowledge needs to be entered about the number of times to unroll loops. Note that this analysis is "anytime": At any point in time, the analysis can be paused, and information can be read out that correctly describes the program's runtime behaviour.

Algorithm 6.1 Applying state joining to a program

```
Require: A $CFG$ and an initial symbolic state $s_0$
1: $Fringe \leftarrow \{s_0\}$
2: while some $s \in Fringe$ has $ip(s) \neq IP_{final}$ do
3:     for all $\{s_1, s_2\} \subseteq Fringe$ such that $ip(s_1) = ip(s_2)$ do
4:         $Fringe \leftarrow \{s_1 \sqcup s_2\} \cup (Fringe \setminus \{s_1, s_2\})$
5:     end for
6:     $s \leftarrow \textsc{choose}(Fringe)$
7:     $Fringe \leftarrow (Fringe \setminus \{s\}) \cup \textsc{execute}(s)$
8: end while
9: return $s$ where $\{s\} = Fringe$
```

6.4 The Algorithm

The algorithm (Algorithm 6.1) takes a program to analyse, in the form of a CFG, and the initial state comprising the PC 1, the IP $IP_{initial}$, and symbolic variables for all inputs. The symbolic execution will then produce results covering all possible values of these symbolic variables. Note that, because of the state joining at Line 4, at Line 6, at most one element of *Fringe* will correspond to IP_{final}. Therefore, at the termination of the algorithm, *Fringe* will be a singleton set whose element corresponds to IP_{final}. The algorithm presented is somewhat simplified, in that it does not show the decompilation of the program needed because we are analysing binaries. In practice, we interleave decompilation with the symbolic execution.

Algorithm 6.1 relies on three key functions not defined here: *execute*, *choose*, and $\sqcup$ (join on states). The *execute* operation extends the supplied state by executing the instruction at its IP. Note that this will produce more than one state for selection (conditional or computed branch) instructions. The next sections describe the *choose* and $\sqcup$ functions.

6.4.1 Preparing to Join

We want to propagate states, and then stop them where they can be joined. Figure 6.8 shows two fragments of control-flow graphs. On the left, state m's instruction pointer is two blocks away from a join point, while state n's instruction pointer is one block away from the same point. We wish to execute n for one block, and m for two blocks. The two states could then be joined. If either state is run too far, past the same-IP point, that chance to join states is lost.

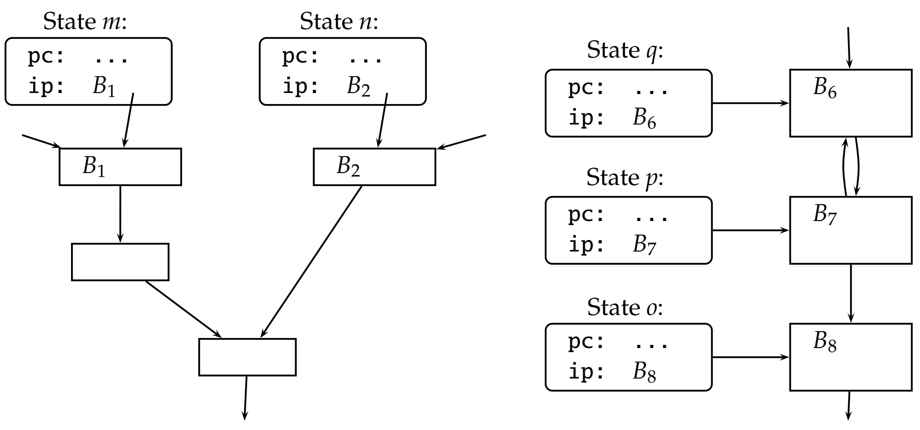


Figure 6.8: CFG fragments. State m's IP is two blocks from a join point. State n's IP is one block away.

For each state, we find all of the descendants of that state in the partial CFG. Because every path finishes at the same exit block, the paths from the states will intersect. For each state we find the minimum distance of that state to its earliest descendant that is common to another state—we call these join points. We find for each state the minimum number of edges that can be traversed before a join point is reached. For each state we now have the minimum distance to its next join point.

Next, we remove from consideration as the next state to run, any state that post-dominates another state. A node o post-dominates another node p if all paths from p to the exit node must pass through o. Since the post-dominated node p should pass through the dominator o, we do not wish to execute the post-dominating state.

The right of Figure 6.8 shows an example where the post-dominance check applies. Each node is zero distance from its earliest join point. That is, if each node is not advanced, another state may be joined with it. In the figure if state o were advanced, it would be moved away from its join with states q and p.

We then choose a state to run for as many blocks as it is from its nearest join point. This is not perfect, for instance a control transfer instruction might visit a new block, causing us to transition through and miss a join point. We build the control flow graph from all the control flow transfers that have occurred so far in the simulator, as well as the static jumps—that is, jumps to a fixed location, where that location has already been disassembled. Runtime calculated jumps (such as returns from functions) that we have not yet seen are omitted. When control transfers to a new location, we perform a dynamic disassembly and incorporate the resulting blocks into the control flow graph. Note that, if the nearest join point is missed, this does not compromise correctness.

6.4.2 Joining and Splitting

States are joined, and sometimes split. A join occurs when two states will execute the same instruction next. A split occurs when the location of the next address to execute depends on a symbolic expression. We split in two contexts. Firstly, on reaching a conditional branch instruction whose condition φ depends on symbolic values, the current PC is conjoined with φ, and separately with $\neg\varphi$. If both are satisfiable, the state is split and two states are created.

The second context involves states that have previously been joined. When a function call may return to multiple locations because the return address was joined from states that called the function at different locations, the state needs to be split at the `return` statement of the function to return the respective states back to their call sites. We create a new state for each distinct next instruction. For example, if the instruction pointer can be 2 or 4, then we split the state, resulting in two states, one with a PC of $(PC \land IP = 2)$ and another with a PC of $(PC \land IP = 4)$.

To join states s_1 and s_2 where s_k contains PC_k and register and memory locations $loc[i]$, create a new state s with $PC_s = PC_1 \lor PC_2$, and for all i, $loc_s[i] = ITE(PC_1, loc_1[i], loc_2[i])$. That is, if PC_1 is 1, use the value from the first state, otherwise use the value from the second state. This is allowable because PCs are always disjoint.

6.5 Simplifications and Approximations

Symbolic execution, even of small programs, can result in large symbolic expressions. This is especially so when analysing machine code. The symbolic expressions that are built are structurally hashed (section 2.6), but even so become large. Every machine instruction which is executed may create a new symbolic expression.

We also apply range and domain analysis of pointers to reduce the number of solver calls. The approximations we use over-approximate the encoding of the constraints, but those constraints describe precisely (without approximation) the behaviour of the program.

6.5.1 Path Constraint Simplification

The Multiply example (Figure 6.5) has a simple branching structure, making its PC easy to simplify. However, programs with more complicated control flow, emanating, for example, from `break` statements, can benefit from PC simplification. Without simplification, the PC becomes large.

In this section we simplify the PC by abstracting its primitive constraints as propositional variables. We then apply Boolean simplifications to reduce the number of propositional variables in the PC, hopefully making the PC easier to handle for a theory solver. We use a, b, and c to refer to propositional variables that describe individual `QF_ABV` constraints.

Continuing with the Multiply example, consider point 4 in Figure 6.6, where the $(x \neq 0 \wedge even(x))$ state joins with the $(x \neq 0 \wedge \neg even(x))$ state. Letting $a = (x \neq 0)$, and $b = even(x)$, the joined PC becomes $(a \wedge b) \vee (a \wedge \neg b)$. Applying the obvious simplification reduces the joined state's PC to $(x \neq 0)$. Heuristic DNF minimisation tools that apply such rules are available; we use Espresso [Rud86].

State splitting complicates this minimisation. Consider for example a **return** statement from a function that returns to one of three addresses; perhaps because calls from three different sites were joined. Let the potential new IP addresses be 4, 8 and 12, let the PC before the split be PC_0, and let IP be the symbolic expression for the instruction pointer when the transfer occurs. Then after the split there will be three states: $(PC_0 \wedge IP = 4)$, $(PC_0 \wedge IP = 8)$ and $(PC_0 \wedge IP = 12)$. If all three states are later joined, the PC will become $PC_0 \wedge (IP = 4 \vee IP = 8 \vee IP = 12)$ which should be simplified to PC_0. A Boolean minimisation algorithm will not do so—because the second disjunction is not obviously true. We can, however, assist the minimisation algorithm by modifying the constraints. Let $a = (IP = 4 \vee IP = 8)$, and $b = (IP = 4)$. Three equivalent PCs that can easily be minimised are: $(PC_0 \wedge a \wedge b)$, $(PC_0 \wedge a \wedge \neg b)$, $(PC_0 \wedge \neg a)$.

Removing tautologies from the PC helps simplification. For example, consider a PC that contains both $x = 0$, and $\neg(x \neq 0)$. Label them a and b. No state will have the PC $(a \wedge \neg b)$ that can simplify the b term. It is safe to remove constraints entailed by other conjoined constraints, these constraints subsume prior constraints—they add no information.

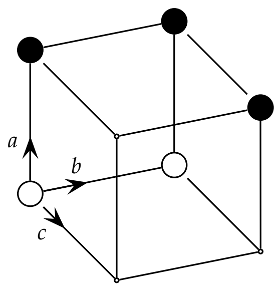


Figure 6.9: Two path constraints

State joining produces many ITE expressions, so we carefully simplify their guards. We calculate four possible guards and, as we describe, use the smallest one. Consider two states to be joined, with $PC_1 = (\neg a \wedge \neg c)$ and $PC_2 = ((a \wedge \neg c) \vee (a \wedge b \wedge c))$. The PCs are shown on Figure 6.9's unit cube—hollow circles for PC_1 and filled circles for PC_2. One possible ITE to use for the joined locations is: $loc_{new}[i] = ITE((\neg a \wedge \neg c), loc_1[i], loc_2[i])$. Another reasonable choice for an ITE is to take PC_2 as the guard, and swap the order of the remaining arguments. However it may be possible to generate a smaller guard by considering that the ITE expression will only be evaluated if $PC_1 \vee PC_2$ is 1. Inspection of the cube shows that the potentially simpler guard of a is equivalent to $a \wedge \neg c$. a covers only vertices of PC_1 and those that we don't care about, it covers none of PC_2's vertices.

To minimise the guard, we mark the vertices of PC_1 as 1, those of PC_2 as 0, and the rest as don't care. Then we minimise using Espresso. Then we swap, marking PC_2's vertices 1, PC_1's 0, and we minimise again.

As guard we choose the expression with the smallest number of nodes in its `QF_ABV` representation. These are candidate guards: PC_1, PC_2, the restriction of PC_1 to $PC_1 \vee PC_2$, and the restriction of PC_2 to $PC_1 \vee PC_2$.

During the constructions of symbolic expressions we follow standard practice and apply rewriting rules to simplify expressions, for example turning $x + 0$ into x.

6.5.2 Value Analysis for Pointers

In the `atol()` function used by a later example (Figure 6.10), each character is looked up in an array to determine if it is a digit. So there is a lookup: $isDigit[c]$, where $isDigit$ is an array of 256 values indicating whether or not each c is a digit. We analyse such pointer accesses using three techniques: first by analysing the domain of the expression; if that fails, by analysing the range of the expression; and if that fails, by solving for each memory address in turn.

The *isDigit*[*c*] expression will translate into a symbolic memory access such as $(base + (c \ll 1))$, that is, access the memory location at the value of the character times two plus some fixed base (the location in memory of the zero index). As this expression contains only a single 8-bit variable, it can take on only 256 possible values. Because the domain is small, we build an expression using each of the possible concrete values, ignoring the PC.

If the domain of an expression is large, we use a bit-vector interval analysis [War02] of the expression to calculate the interval of the range. We use a signed interval, where the lower bound is smaller than the upper bound. For example, a subtraction expression over two 8-bit values each with a range of [0,1] has a resulting range of [-1,1]. We widen to the whole range when an overflow occurs. A signed range allows us to handle an expression such as $(2 \times t^{[8]}) - 1$. If $t^{[8]}$ is in [0,12], then the range of the expression is in the signed interval [-1,23]; to be correct, an unsigned interval would widen terribly to [1,255]. Because of the ubiquity of aligned memory accesses we use strides, allowing regular gaps in the interval, for example a stride of 2 on the interval [0,6] contains [0,2,4,6]. When performing the interval analysis, we likewise ignore the PC. Balakrishnan [Bal07] gives the functions to calculate the precise ranges for logical and arithmetic operations for strided intervals. Navas et al. [NSSS12] use intervals that gracefully handle overflow, maintaining interval precision even in the presence of overflow.

If the domain and range analysis produce too many values, or if some of the values are not addressable, we (inefficiently) solve iteratively for the index subject to the PC. For example, given a symbolic expression *index*, which first solves to 2, we then solve *index* such that $index \neq 2$, and so on. The solvers we use do not have the option to produce all the satisfying assignments to a formula—which we require to check all possible memory addresses. If there are many possible memory addresses, then this operation will be slow.

6.6 MinkeyRink's Implementation

We analyse Linux x86 executables, both compiled binaries and interpreted scripts. Given an executable, and an input width, we dynamically disassemble the executable as it runs on a random input of specified length. The result of this dynamic disassembly is a trace through the program. MinkeyRink then symbolically executes using the control flow graph derived from the trace. When a control transfer instruction is reached which depends on a symbolic variable, the bit-vector/array solver is called and the state split if necessary. When symbolic execution reaches an address that has not already been disassembled, a new dynamic disassembly is triggered. The resulting trace is merged with previous disassembled traces to build a more precise control flow graph. We follow Minato [Min96] and check the satisfiability of each guard. A loop is unrolled until its guard evaluates to 0.

We use the Valgrind framework [NS07] to disassemble. Valgrind is a widely used dynamic instrumentation framework for analysing x86 and PowerPC executables on the Linux and AIX operating systems. We use Valgrind to identify instructions, and also because it translates x86 instructions into a more manageable RISC language. There are three advantages of dynamic disassembly over static disassembly. First, during disassembly the results from system calls are collected; these are later played back to the simulated program. Second, we disassemble the dynamically linked libraries no differently from the program's instructions. Third, interpreted scripts, and their interpreters, can be analysed. Our dynamic disassembly is time consuming, others [NLLC06] have used an initial static disassembly to save time.

The PC and symbolic expressions are stored inside MinkeyRink as Valgrind expression graphs, and converted to the `QF_ABV` language for solving. We do not utilise the solver's custom interface. Instead we generate standard SMT-LIB format and communicate via temporary files. Using the standard interface requires the solver to redo work, but allows easy experimentation with different solvers.

We have built a simulator of Valgrind's instructions, which executes the instructions symbolically and concretely. It simulates many, but not all, of Valgrind's instructions. Currently, MinkeyRink does not handle multi-threaded programs, asynchronous signals, or floating point instructions.

```
int main(void) {
    char s[9];
    long number;

    printf("Enter a Number:");
    fgets(s, sizeof(s), stdin);
    number = atol(s);
    if (number == 12345678) {
        fail();
    }
    return 0;
```

Figure 6.10: Number: Error on input 12345678

```
uint64_t popCount (uint64_t y) {
  uint64_t c;
  for (c = 0; y; c++)
    y &= y-1;

  return c;
}
```

Figure 6.11: Wegner: Counting 1-bits

The program's state is modified by Linux system calls. We have symbolic versions of some system calls only, the remainder we replay from the traces captured during dynamic disassembly.

6.7 Results

To explore the usefulness of the state joining approach we analyse four programs. One is a simple C program that will fail if a particular integer is input (Figure 6.10). In this example, the function describing the result of `atol()` is surprisingly complicated: Given 9 characters of input, there are about 10^{20} different inputs that will evaluate to 1, corresponding to white space followed by an optional plus sign, followed by 1, followed by non-numeric characters. The second is a program that counts the number of 1-bits in a number (Figure 6.11). The third is the Multiply example of Figure 6.5. The fourth example is the gzip file compression program.

In the Number example, an input that causes the program to fail is derived. For the remaining three programs the input-output function is derived. For the Multiply

	Problem			
Category	Number	Gzip	Wegner	Multiply
Bytes input	9	1	8	16
Dynamic Disassemblies	9-16	1	1	1
With joining				
Total time	126s	> 30000s	33s	45s
Solver time (STP r60)	90s	> 26973s	28s	11s
Solver calls	110	> 684	67	131
Boolean simplification	4.5s	> 103s	3s	5.5s
Joins	60	> 510	64	127
Maximum height	94	> 500	67	131
Maximum width	5	2	2	3
Without joining				
Total time	732s	4006s	35s	–
Solver time	555s	3312s	25s	–
Solver calls	6929	34944	66	–
Boolean simplification	82s	52s	3s	–
Paths for Symbolic Exec.	1662	256	65	2^{64}

Table 6.1: Results of applying state joining. Running with STP revision 60. Gzip with state joining timed out after 30,000 seconds.

example, this could be used to show the commutativity of multiplication. For the Wegner example [Weg60], this could be used to determine that the return value is always less than 65. For gzip, this could be used as the first part of establishing that gunzip composed with gzip is the identity function for some size inputs.

Table 6.1 shows the results of running these four programs. Some of the rows are: *Dynamic Disassemblies*: the number of calls to the dynamic disassembler (the value varies depending on the initial random input chosen); *Joins*: the number of pairs of states that were joined; *Maximum Width*: the greatest number of active states at any time (the maximum width of the state graph); *Maximum Height*: the longest path through the state graph.

We made three runs of each analysis; the times shown in Table 6.1 are arithmetic averages. Times were measured on a single core of a Pentium D 3GHz, running Ubuntu 8.04. We use revision 60 of STP, which we found to be the fastest available solver at the time.

A program that generates all one byte files, then gzips them, then builds an input-output function takes 7 seconds to run. To produce the equivalent formula by symbolic execution takes 4006 seconds, and to produce the same formula by state joining timed out after 30,000 seconds.

Gzip, when symbolically executed, produces singleton states—each input follows a different path. So symbolic execution has no advantage over dynamic analysis. The PC that state joining produces is more difficult to solve, overwhelming the savings from merging.

For the Wegner example, the number of paths is equal to the bitwidth plus one; in our example, 65 paths. State joining on this example joins just before the exit—the same as symbolic execution. The Wegner example benefits neither from the Boolean simplifications nor from the pointer value analysis. The Boolean simplifications occur at the join point at the function's `return`, when the PC is no longer used. The pointer value analysis, as for the Multiply example, does not help because the analysis produces no symbolic pointers.

The Multiply example has a simple structure, with control transfer instructions that join back on each other—producing constraints that easily cancel out. With Boolean simplifications disabled, the example takes more than 50 times longer to run. It is well suited to state joining. Note that we pass the function symbolic variables, not characters turned into numbers. Parsing the numbers would require effort comparable to that in the Number example.

6.8 MinkeyRink with STP2

When we performed the evaluation of MinkeyRink, before we began work on STP2; STP r60 was the best available bit-vector and array solver for the problems we generated. This is a version of STP prior to the improvements described earlier in this dissertation. In this section, we re-run some of the same test problems shown in Table 6.1 with STP2 r1654. The results in Table 6.2 show the time spent solving on a single core of a Pentium D 3GHz, running Ubuntu 9.04. The table gives just the time spent in STP2. Again, we did not attempt to run the Multiply example without state joining because of the very large number of paths.

We have not been able to get MinkeyRink analysing gzip reliably on Ubuntu 9.04. Since we did the initial work, vector instructions have been compiled into the standard libraries used by gzip, which MinkeyRink does not currently analyse.

Problem	Solver calls	STP r60	STP r1654
With joining			
Number	179	101s	39s
Wegner	67	30s	14s
Multiply	131	12s	17s
Gzip	–	–	–
Without joining			
Number	3697	375s	200s
Wegner	66	27s	13s
Multiply	2^{64}	–	–
Gzip	34944	2031s	2878s

Table 6.2: Results of applying state joining, comparing STP r60 and STP r1654.

However, we stored the longest running `QF_ABV` problem we encountered during the evaluation (Table 6.1). STP r60 takes 2862 seconds to solve this problem. STP2 r1661 solves the same problem on the same computer in just 26 seconds.

The results in Table 6.2 show only a modest improvement for STP2 versus STP. Many of the problems are small and easy. The more advanced simplifications that STP2 performs have an overhead that is not justified for such easy problems.

6.9 Complications

There are some practical complications with state joining for executables. Linux has hundreds of system calls that can modify the program's state. MinkeyRink has symbolic versions of the semantics for only a few system calls; for the remainder we replay the system call's results which Valgrind captured during the dynamic disassembly. Before replaying the result of a system call we check that the parameters are the same. Our assumption is that system calls that are called in the same order with the same input will produce the same results. This limits further the strength of the guarantee we extract. For example, if a program would behave differently on different dates, we would not discover this, as the result of the system call that returns the date is not made symbolic.

Because we replay traces of the system calls, if inputs cause the program to make different sequences of system calls, the analysis will not have the appropriate system call to replay. One solution may be to split the state whenever the sequence of system calls changes, but we do not do this yet.

```
if (a>0)
    p = malloc(100000);
else
    p = malloc(0);
*p = 0;
```

Figure 6.12: A complication for state joining

If two states have different system call traces then they might not be appropriate to join. At present we cannot analyse the program shown in Figure 6.12, as the memory assigned on each branch is different. If we allowed the memory mapping system calls to vary on branches then one state would have memory allocated that the other did not. States cannot be joined just when their next instructions are the same. The system calls to the operating system that have been performed, such as allocating memory, and opening files need to be checked when joining states. We only join states that have performed system calls we know are safe. For instance, we do not join states that have different files open, or different memory allocated.

Using dynamic disassembly, we cannot visit locations that are not reached at runtime, so we cannot analyse error handling code unless the error occurs at runtime. For example, if we wish to insert error return values from system calls, such as a file read failing, then we need to disassemble the error handling code. Currently we cannot introduce failures when performing a disassembly, so we cannot explore the error handling code.

Over-zealous joining is detrimental. For small functions, joining is undesirable, as the cost of joining/splitting overwhelms the saving. If a function is called from different sites and contains few instructions, the joined states will run for a few instructions before splitting when the function returns to the different call sites. We need heuristics to decide when to join.

Another limitation is that symbolic execution generally operates on a fixed input width, say 20 bits. Depending on the program structure, greater input lengths may be required to cause a particular failure. Symbolic execution builds expressions that describe the program for some fixed length input. For some inputs this may be equivalent to checking each smaller length input, but usually not.

When choosing what to join, we do not consider the call site. Consider a function which is called at the start and end of a program. Our analysis does not take the call site into consideration, so it believes that calling the function could return to either the start or end of the program. Our analysis is context insensitive: it does not determine that the return address on the stack is the particular address that will be returned to.

6.10 Related Work

There has been a tremendous amount of software verification research. In this section we focus just on closely related work. Bounded model checking and abstract interpretation are the two most popular competing alternatives to symbolic execution.

The idea of symbolic execution (MinkeyRink's conceptual basis), is usually attributed to King [Kin76], but others such as Howden [How77] and Clarke [Cla76] were working on the approach at the same time.

From the early 1980s until about 2000, little research was conducted into the symbolic execution of software. By 2000, enough had changed to make symbolic execution of software practical. First, owing to the rise of the internet, we saw more security critical programs; many more programs were exposed to untrusted inputs, for instance web browsers, document readers, and media players. Second, the size of the programs we wish to analyse has grown slower than the speed of computers. Now, we are interested primarily in the parts of the programs that read and verify the structure of untrusted input. Third, Boolean satisfiability solvers are faster than ever.

The most obvious change to automated test generation (ATG) systems over the last 30 years has been that the constraint solving engines available to ATG systems have become more powerful. SELECT [Kin76] solved problems using a linear and conjugate gradient solver. Clarke analysed FORTRAN using linear solver, while EFFIGY used a polynomial solver. Systems like EXE [CGP$^+$06] use SMT bit-vector and array solvers.

SE tools need to mark some values as symbolic. This is typically done by marking input from the user, file or network as symbolic variables. However, it could be any values, even the intermediate values in computations. In our work we mark input as symbolic. Godefroid et al. [GKL08], instead of marking the input as symbolic, mark the result of the lexer as symbolic. Marking the lexer's result as symbolic avoids redundantly generating input that the lexer considers to be identical, for example multiple white space characters. Wherever in the program the symbolic variables are introduced—the tools and theory are the same.

6.10.1 Tools

Currie et al. [CHR00] use SE to show equivalence between optimised and unoptimised sections of assembly code for a digital signal processor.

Larson and Austin [LA03] track the intervals that variables may range over in a symbolic state. They use the analysis to find accesses to out of bounds memory. Their tool does not reason precisely about overflow. Cadar and Engler [CE05] describe EGT, a structural fuzzer which uses source-to-source transformation, and the CVCL SMT solver. Sen et al. [SMA05] describe CUTE, a tool which operates on the source code representation and uses linear constraints in the symbolic state.

EXE [CGP+06], and its successor Klee [CDE08], starts from an actual execution, then negates each constraint of the path constraint one-by-one. A new execution path is generated that takes the same path up until the inverted constraint, but afterwards it takes a different branch. Paths that will visit previously un-reached locations are prioritised. EXE, with its aim of visiting previously un-reached statements, has heuristics that produce statement coverage. These heuristics are much simpler than attempting to derive input that reaches a particular instruction. In practice they achieve good results.

SAGE [GLM08] likewise negates constraints one-by-one and solves them, but prioritises the PCs that visited new states. The SAGE system starts from the first constraint in the PC, and negates it. If the prior constraints conjoined with the negated constraint is satisfiable, then a new input has been found. The program is run dynamically with that new input. Inputs are scored based on the number of new blocks they reach. The PC of inputs that discover more new blocks are scheduled before those that discover fewer.

EXE and SAGE are both defect-finding tools; they do not exhaustively check each path. Their heuristic of preferring paths that visit previously unvisited statements prevents the path search from unrolling the same loop indefinitely. Neither tool performs a directed search for problems. Instead, both tools happen across defects as they explore paths.

Other approaches have been developed to target particular parts of a program for execution, for instance Ma et al. [MPFH11]. Several other structural fuzzers for binaries have been developed, for instance OSMOSE [BH11] and Avalanche [IS10].

6.10.2 Symbolic Memory Accesses

Symbolic memory accesses are generally handled in one of three ways: by concretisation, exhaustion, or abstraction. Concretisation dramatically increases the number of paths through the program. Handling the accesses exhaustively may require large arrays to be included in the formula. For example, if a symbolic index could take on 2^{12} values then all 2^{12} values need to be encoded in the formula. Abstraction of arrays is often performed by first setting the result of the memory access to an unconstrained value, that is, any possible value. The precise expression is included only if needed.

SAGE concretises symbolic index expressions. Coen-Porisini and De Paoli [CPdP93] employ the exhaustive approach, associating with each symbolic variable a set of symbolic values and the predicates for when the symbolic value applied. They provide a denotational semantics for the approach, but not a description of how to compress the constraint. King [Kin76] forks execution on each constraint value, in effect checking concretisations one by one, which we found to be too slow.

EXE divides memory into regions; this allows the solver to work on different distinct parts of memory separately. It determines which region the concrete pointer indexes into, and constrains the symbolic value to be inside this region. This is similar to solving the bounds to establish the range of the pointer.

The abstraction approach to handling symbolic index expressions is used in a different setting by Engler and Dunbar [ED07]. Under-constrained symbolic variables are variables that do not have all the appropriate constraints applied to them. If an under-constrained variable causes a failure, an appropriate constraint (such as that the denominator is not zero) is applied to it, and execution continues if the constraint system is satisfiable. If the constraint is unsatisfiable then the failure must occur. This is unsound, but useful in their setting, for testing subsystems.

6.10.3 Handing the Path Explosion

The path explosion problem of symbolic execution has been addressed by others. Kölbl and Pixley [KP05] investigate state joining of programs written in a subset of C++, and describe it well. The principal difference with our work is that we focus on analysing arbitrary binaries which can use dynamic memory and pointer arithmetic.

Boonstoppel et al. [BCE08] discard states that differ from other states only in locations that will not later be read. Consider a conditional output statement such as `if (guard){printf(‘‘value’’);}`. If both branches are taken and outputs are ignored, the states do not differ. One can be discarded, as the remainder of their paths will be the same. This approach requires the calculation of which locations will be read and written to in the remainder of the path. Deciding this statically for machine code is more difficult owing to the more complicated control flow transfers. Our approach is more general—allowing the joining of states that differ in variables, while not requiring the calculation of which variables may be read from or written to later. The calculation does allow discarding of symbolic expressions that will not be used later—a good way to conserve memory.

Kuznetsov et al. [KKBC12] estimate the effect of merging states based on how symbolic variables are later used. They show promising results.

6.10.4 Other

To automatically vectorise loops during compilation, Allen et al. [AKPW83] use Boolean simplification and *if-conversion* to remove if-then-else statements. A statement such as `if(g){y=x} else {y=z}` is converted into:

`y = (x & guard | y & guard)`, where *guard* is the sign extension of g to the bit-width of y, where g is considered to be a 1-bit variable. We suspect that compilers, like GCC, which implement if-conversion will produce binaries that MinkeyRink can analyse more efficiently, because fewer states are split.

Arons et al. [AEO$^+$08] fork execution at each branch, adding both paths to a list of paths to explore. If paths are stopped at the same location, they are merged. The merging is performed like we do, effectively creating an if-then-else where the guard is the respective path-constraint. The merging is sensitive to the order of path exploration; only paths currently stopped at that location are merged. The approach, while good, relies on users choosing good merge points. The approach that we develop is completely automatic.

The Calysto tool [BH08] also merges paths, but only unrolls loops once. Calysto symbolically executes the program's functions, so each function is only analysed once. Calysto then inlines each function at call sites. In our case, unrolling the loop once only causes an unacceptable loss in precision, probably because we analyse at a binary level where each iteration does less. Calysto, on the other hand, achieved good results with the once-only strategy.

Symbolic execution has also been used to show the equivalence of programs [Min96]. Minato's system [Min96] handles conditional branches (if-then-else) and data dependent loops (while-end) using BDDs. Minato demonstrates this on Euclid's GCD algorithm, producing a BDD that encodes the function for up to two 10-bit inputs. Conditional branches are transformed statically to the equivalent ITE functions. So `if (a)then {b=c} else {b=d}` becomes $b \leftarrow ITE(a, c, d)$. Loops are handled similarly: new ITE expressions are introduced until the BDD that encodes the guard is unsatisfiable. So after two passes, the loop `while(a){b=c}` is transformed to $b \leftarrow ITE(a_2 \wedge a_1, c_2, ITE(a_1, c_1, c_0))$, where the subscripts refer to the value of the variable at that iteration.

Godefroid [God07] describes a compositional analysis. When a function is called, the changes that the function makes are recorded, and the PCs added by the function are recorded. Later if the function is called again, the PC is checked for satisfiability; if it is satisfiable, then the results from the prior function call are written into the state. If the PC is unsatisfiable, then the function is executed, and the PC and results stored. With enough calls, the disjunction of PCs from the executions will cover any possible input. This system differs from Calysto in that the results of functions are composed run-by-run. The stated advantage of this demand-driven approach is that extra unfeasible paths are not summarised. With machine code, this approach is more difficult to apply. There could be differences between the calling contexts of a function that are not obvious, especially when analysing machine code, where the operands of functions are not explicitly indicated.

Clarke et al. [EC03] unroll loops and install an unwinding assertion. Loops are unrolled, up until a limit, until the unwinding assertion is necessarily 0. They apply

```
if(e)
{
   I();
   if (e)
   {
      I();
      if (e)
      {
        I();
        assert(!e);
      }
   }
}
```

Figure 6.13: Example of unrolling 3 times. The loop is unrolled until the unwinding assertion, shown here as a n “assert” is necessarily false, or until an unwinding limit is reached.

source transformations to reduce the control transfer instructions to just `goto` and `if`. For example, unrolling `while(e){I();}` three times produces Figure 6.13.

Xie and Aiken [XA05] unroll loops using BDDs to associate guards with each statement. They contribute a programming language semantics that details formally how to translate operations into updates of the guards and states. One state subsumes another if its state is a superset of the other state. For example, two paths through a program that differ in what they output could be joined if the output makes no difference to the state.

Boonstoppel et al. [BCE08] perform a liveness analysis on symbolic expressions, pruning those that are unreachable from the path constraint, and then merging paths. Their insight is that multiple paths often produce the same effect, either if there are no side effects from branches, or after the side effects have operated. For example, at the end of a block, one path may have the constraint that $\{c \neq 0\}$ and the other that $\{c = 0\}$. If c is dead, however, this distinction is immaterial. Note that the properties need to be checked before merging, in case the path difference triggered a fault.

6.11 Conclusion

State joining as we have implemented it has varying performance. The performance of the approach depends on the difficulty of solving the generated constraints. On the gzip example, the constraints became so expensive to solve that state joining was slower than both exhaustively executing the program and symbolically executing it. In particular, gzip produced many symbolic memory indexes which slowed down STP r60.

However, as SMT bit-vector and array solvers become more sophisticated, approaches like the one we have described will become more practical. Over time we hope that the underlying solvers will increase in sophistication, removing the need for tools to carefully manipulate the problems they need solved.

Incidental to deriving the program's input-output function, we extract an accurate (partial) CFG from the binary code. The approach generates a safe under-approximation of the CFG using a flow sensitive analysis. A corresponding upper-bound of the CFG can be produced by abstract interpretation [KVZ09].

State joining is useful if the following conditions apply: the paths call a similar sequence of system calls, the number of paths through the program is large, and memory is rarely written to at symbolic locations.

Three improvements to our implementation are apparent. First, it is common around loops for later constraints to imply earlier ones. It is not apparent to a propositional simplifier that (from the Multiply example): $(x_0 \gg_l 2) \neq 0$ implies $(x_0 \gg_l 1) \neq 0$. Removing earlier constraints when they are implied by later constraints is desirable because it reduces the redundancy. Second, and related, is that with our current simplification scheme based on propositional variables, the performance of the analysis is dependent on whether the constraints simplify during joining. If the joined PC can be simplified, as in the Multiply example, performance is good. However, slight syntactic changes to conditionals can dramatically increase running time. For instance, changing the Multiply example slightly so that the diamond shape is lost, causes analysis to take more than 100 times longer. Using the solver's native interface to maintain the state's PC would reduce the amount of work the solver needs to perform. Third, the use of a generalised memoization (compositional) would reduce the amount of re-work performed. Currently we reprocess functions repeatedly rather than reusing the prior work that was performed.

Normal symbolic execution of binaries allows arbitrary properties about the input-output function of programs to be verified, but the technique works poorly on programs that have many paths through them. We have investigated how state joining may help. So far we have a number of promising results for analysing unmodified executables, as well as examples that do not benefit. However, this is an area full of opportunities for future improvement.

7 Conclusion

THIS thesis has investigated building bit-vector and array solvers, and their application to analysing machine code programs. Bit-vector and array solvers are widely used to answer questions about the behaviour of software. Analysing machine code programs is a way to automatically discover low-level defects, like division by zero.

In our investigation of bit-vector solvers we largely focussed on simplifications—ways to make bit-vector problems easier to solve. We now summarise our contributions to simplification.

Our variable elimination approach (section 3.4) is a conceptually simple way of repeatedly isolating variables on one side of an equality. It is more general than other approaches because, for instance, it also eliminates variables that occur in bit-vector xors.

Bit-blasting equivalence checking (section 3.8) transfers equivalences detected by and-inverter graphs (AIGs) back to the bit-vector theory-level. The equivalences that are deduced during bit-blasting are used to further simplify the bit-vector problem.

A new approach to discovering equivalences (subsection 3.11.1) provides a way for authors of bit-vector solvers to discover new and interesting rewrite rules. Automatically discovering equivalences makes it less likely that useful rules will be missed. It helped us find rules that we would otherwise not have discovered.

Theory-level bit propagation (chapter 4) simplifies bit-vector problems by deducing the value of some bits at the bit-vector level, rather than at the propositional level. For each operation in QF_BV we built a propagator which transferred bit information between its operands and result. We found that Z3, when enhanced by this approach, answered 10% more of the test problems.

Comparing each propagator against the result of the corresponding optimal propagator (section 4.8) showed that the implementation of several of the propagators was optimal on all the assignments generated. Likewise, on all the assignments generated, unit propagation over the CNF of bit-vector xor, bit-vector or, bit-vector and, and equals were optimal. If building a combined SAT and propagation solver, it would make sense to encode those operations as CNF rather than using propagators for them because their CNF encodings are compact and propagate strongly. For multiplication we applied a novel technique which we called column bounds propagation, which subsumes other more simple propagators.

A common reason not to build optimal propagators is that the advantage of the extra precision is outweighed by the extra time it takes to obtain the precision. Measuring the running time of the propagators (section 4.10) showed that generally the implementations are efficient. However, the implementation of the 6-bit optimal multiplication propagator (section 4.9) was too slow to be useful. We found it useful to calculate the effect of the optimal propagator without undertaking the effort to build the propagators. We determined that theory level bit propagation would provide a benefit before having invested the effort to build the propagators.

The $\mathcal{DCI}$ array solver (chapter 5) includes a lazy approach to clause generation. Compared to an abstraction-refinement solver, it asserts clauses to the SAT solver sooner after they are required. The $\mathcal{DCI}$ solver works particularly well on problems that require many abstraction-refinement iterations. However, the disadvantage of the $\mathcal{DCI}$ approach is that the implementation is tied closely to a particular SAT solver, in our case Minisat 2.2. Implementing the approach in a more modern SAT solver is complicated because of the complex invariants that need to be maintained. For instance, Lingeling ([Bie12]) maintains several data structures relating to the clauses that need to be incrementally updated as extra clauses are added asserted during the search. Other approaches (like $\mathcal{A}ck$) have better modularity, making it easy to use whichever SAT solver is the best or most appropriate at a given time.

The $\mathcal{DCI}$ array solver was slower on the evaluation problems than another much simpler array solver ($\mathcal{A}ck$). On test problems STP was about 7 times faster than a prior STP version (subsection 5.6.5).

A goal of some abstraction-refinement array solvers is to instantiate a small number of function congruence constraints each iteration. In the worst case these solvers perform $O(k^2)$ iterations, where k is the number of array selects. STP 0.1 has a lower upper limit of k, asserting all of the function congruence constraints when the limit is reached—this can save considerable time.

MinkeyRink (chapter 6) is a tool for analysing binary programs. MinkeyRink joins and splits symbolic states to overcome the path-explosion problem of symbolic execution. Joining states means that an exponential blow-up in the number of states can sometimes be avoided. Our implementation was sensitive to the order in which states were joined. If states were joined so that the path constraint simplified nicely then it was effective. The tool could be improved by taking more care when choosing which states to run so that the path constraint simplified more often.

Some of the research we have presented, we believe, merits further research.

Technology mapping was the most effective of the solver's phases. However, the comparison we performed compared technology mapping to the Tseitin transformation. A more relevant comparison, which we leave for future work, is to compare Technology Mapping to more modern encodings like the Plaisted and Greenbaum translation. Further, we performed parameter optimisation to amongst other things, select good bit-blasted encodings for technology mapping. To make the comparison fairer, all the parameters should be returned.

The encoding of multiplication using sorting networks seems like it should be effective. Future work is required to understand exactly why it is not helpful.

The algorithm we gave for generating rewrite rules may generate more applicable rewrite rules if constants are not generated on the left-hand side. Further, it might be possible to improve the efficiency of the search for new rewrite rules by better caching calculations.

A recurring theme in this thesis has been that different problems solve faster with different simplifications enabled. The light-weight solver 4Simp (section 3.22) outperformed STP2 on our evaluation problems. However, because 4Simp omits standard simplifications it is not able to quickly answer many problems that STP2 finds easy. For instance, problems that require associative reasoning, such as $a \times (b \times c) = (a \times b) \times c$, are solved much faster by STP2.

For easy problems derived from our software verification tool (MinkeyRink), STP 60 was overall faster than STP2. STP2's cost of performing the extra simplifications is not justified for such easy problems.

Selecting a universally good set of parameters for a solver is difficult, as has been observed by others ([dMP12]). For each simplification we have shown, it is easy to craft problems which are very slow with that simplification disabled. However, enabling simplifications has a cost which sometimes overwhelms the benefit—as we showed with our 4Simp solver.

In this thesis we have focussed primarily on solving isolated problems. In practice, tools like our MinkeyRink analysis tool produce a sequence of related `QF_ABV` problems. A promising avenue of future research is to adapt the simplifications to apply to sequences of problems dynamically. Such an approach might apply parameter optimisation to a sample from the sequence of `QF_ABV` problems it receives and adapt appropriately.

The improved performance of bit-vector and array solvers has enabled tools to reason more precisely and efficiently about the effect of programs. The faster solvers become, the more useful the tools are. There are many promising improvements to apply to solvers and tools, with hard work, these will help millions of programmers discover defects in their software.

Bibliography

[Ach07] Tobias Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.

[ACHB11] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium*, February 2011.

[Ack54] W. Ackermann. *Solvable Cases of the Decision Problem*. North-Holland Publishing Company, 1954.

[AEO$^+$08] Tamarah Arons, Elad Elster, Shlomit Ozer, Jonathan Shalev, and Eli Singerman. Efficient symbolic simulation of low level software. In *DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 825–830, 3001 Leuven, Belgium, 2008. European Design and Automation Association.

[AKPW83] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189, New York, NY, USA, 1983. ACM.

[Ard96] Laurent Arditi. BMDs can delay the use of theorem proving for verifying arithmetic assembly instructions. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 34–48, London, UK, 1996. Springer.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

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

[Bac07] Fahiem Bacchus. GAC via unit propagation. In Christian Bessière, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 133–147, Berlin, Heidelberg, 2007. Springer.

[Bal07] Gogul Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI, 2007.

[Ban08] Sorav Bansal. *Peephole Superoptimization*. PhD thesis, Stanford University, 2008.

[BB04] P. Bjesse and A. Boralv. DAG-aware circuit compression for formal verification. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '04, pages 42–49, Washington, DC, USA, 2004. IEEE Computer Society.

[BB06] Robert Brummayer and Armin Biere. Local two-level and-inverter graph minimization without blowup. In *Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS06)*, 2006.

[BB08a] Armin Biere and Robert Brummayer. Consistency checking of all different constraints over bit-vectors within a SAT solver. In *Proceedings of the 2008 Conference on Formal Methods in Computer-Aided Design*, pages 1–4. FMCAD Inc., 2008.

[BB08b] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, pages 6–11, New York, NY, USA, 2008. ACM.

[BB09] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(1-3):165–201, 2009.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207, London, UK, 1999. Springer.

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking path explosion in constraint-based test-generation. In C. R. Ramakrishnan and Jakob Rehof, editors, *International Conference on Tools and Algorithms for the Constructions and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2008.

[BCF^{+}06] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Alessandro Santuari, and Roberto Sebastiani. To Ackermann-ize or not to Ackermann-ize? On efficiently handling uninterpreted function symbols in SMT (EUF $\cup$ T). In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 4246 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2006.

[BDdM^{+}12] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, pages 1–35, 2012.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Annual Design Automation Conference*, DAC '98, pages 522–527, New York, NY, USA, 1998. ACM.

[BFSW11] Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In *Proceedings of the First International Conference on Certified Programs and Proofs*, CPP'11, pages 183–198, Berlin, Heidelberg, 2011. Springer.

[BG00] Mihai Budiu and Seth Copen Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. Technical Report CMU-CS-00-141, Carnegie Mellon University, June 2000.

[BH08] Domagoj Babić and Alan J. Hu. Calysto: Scalable and precise extended static checking. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 211–220, New York, NY, USA, 2008. ACM.

[BH11] Sébastien Bardin and Philippe Herrmann. OSMOSE: Automatic structural testing of executables. *Software Testing, Verification and Reliability*, 21(1):29–54, 2011.

[BHP10] Sébastien Bardin, Philippe Herrmann, and Florian Perroud. An alternative to SAT-based approaches for bit-vectors. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2010.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, 2009.

[Bie12] Armin Biere. Lingeling and friends entering the sat challenge 2012. In Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, and Carsten Sinz, editors, *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2012. ISBN 978-952-10-8106-4.

[BKO$^+$07] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[BM10] Robert K. Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'2010)*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40, 2010.

[BRST08] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare 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.

[Bru09] Robert Brummayer. *Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays*. PhD thesis, Johannes Kepler University, 2009.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard – version 2.0. In *Proceedings of the* 8th *International Workshop on Satisfiability Modulo Theories (SMT '10)*, July 2010. Edinburgh, Scotland.

[BSWG00] Mihai Budiu, Majd Sakr, Kip Walker, and Seth C. Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Proceedings of the EuroPar 2000 European Conference on Parallel Computing*, volume 1900 of *Lecture Notes in Computer Science*, pages 969–979. Springer, 2000.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[CE05] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In Patrice Godefroid, editor, *Proceedings of the 12th International Conference on Model Checking Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23, Berlin, Heidelberg, 2005. Springer-Verlag.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, London, UK, 2000. Springer-Verlag.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322–335, New York, NY, USA, 2006. ACM.

[CHR00] David W. Currie, Alan J. Hu, and Sreeranga P. Rajan. Automatic formal verification of DSP software. In *Proceedings of the 37th Annual Design Automation Conference*, pages 130–135. ACM, 2000.

[Cla76] Lori A. Clarke. A program testing system. In *ACM 76: Proceedings of the Annual Conference*, pages 488–491, New York, NY, USA, 1976. ACM.

[CMR97] David Cyrluk, M. Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, 1997.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC'72)*, pages 151–158, New York, NY, USA, 1971. ACM.

[CPdP93] Alberto Coen-Porisini and Flavio de Paoli. Array representation in symbolic execution. *Computer Languages*, 18(3):197–216, 1993.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[dM11] Leonardo de Moura. What methods does Z3 use to solve quantifier-free bit-vector formulas (QF_BV)? http://www.stackoverflow.com/questions/7268221/, September 2011.

[dMB08a] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, May 2008.

[dMB08b] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[dMB11] Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, September 2011.

[dMP12] Leonardo de Moura and Grant Olney Passmore. The strategy challenge in SMT solving, 2012. unpublished.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75, Berlin, Heidelberg, 2005. Springer.

[EC03] Karen Yorav Edmund Clarke, Daniel Kroening. Behavioral consistency of C and Verilog programs. Technical Report CMU-CS-03-

126, Computer Science Department, School of Computer Science, Carnegie Mellon University, 2003.

[ED07] Dawson Engler and Daniel Dunbar. Under-constrained execution: Making automatic code destruction easy and scalable. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 1–4, New York, NY, USA, 2007. ACM.

[EMS07] Niklas Eén, Alan Mishchenko, and Niklas Sörensson. Applying logic synthesis for speeding up SAT. In João Marques-Silva and Karem A. Sakallah, editors, *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, volume 4501 of *Lecture Notes in Computer Science*, pages 272–286, Berlin, Heidelberg, 2007. Springer.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer-Verlag, 2004.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[Fox11] Anthony C. J. Fox. LCF-style bit-blasting in HOL4. In Marko van Eekelen *et al.*, editor, *Proceedings of the Second International Conference on Interactive Theorem Proving*, volume 6896 of *Lecture Notes in Computer Science*, pages 357–362, Berlin, Heidelberg, 2011. Springer.

[Fra10] Anders Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, University of Trento, 2010.

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

[GBD05] Vijay Ganesh, Segey Berezin, and David L. Dill. A decision procedure for fixed-width bit-vectors. Technical report, Stanford University, 2005.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531, Berlin, Heidelberg, 2007. Springer.

[GKA$^+$11] Vijay Ganesh, Adam Kieżun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael Ernst. HAMPI: A string solver for testing, analysis and vulnerability detection. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, volume 6806 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, Heidelberg, 2011. Springer-Verlag.

[GKL08] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215, New York, NY, USA, 2008. ACM.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *NDSS 2008: Proceedings of the 15th Annual Network & Distributed System Security Symposium*, pages 151–166, 2008.

[God07] Patrice Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[GOSL$^+$12] Vijay Ganesh, Charles W. O'Donnell, Armando Solar-Lezama, Srinivas Devadas, Mate Soos, and Martin Rinard. Lynx: A programmatic SAT solver for the RNA-folding problem. In Alessandro Cimatti and Roberto Sebastiani, editors, *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, volume 7317 of *Lecture Notes in Computer Science*, pages 143–156. Springer-Verlag, 2012.

[HBHH07] Frank Hutter, Domagoj Babić, Holger H. Hoos, and Alan J. Hu. Boosting verification by automatic tuning of decision procedures. In *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '07, pages 27–34, Washington, DC, USA, 2007. IEEE Computer Society.

[HHLBS09] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, September 2009.

[How77] W.E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[HSS09] T. Hansen, P. Schachte, and H. Søndergaard. State joining and splitting for the symbolic execution of binaries. In S. Bensalem and D. A. Peled, editors, *Runtime Verification*, volume 5779 of *Lecture Notes in Computer Science*, pages 76–92. Springer, 2009.

[Hua08] Jinbo Huang. Universal Booleanization of constraint models. In Peter Stuckey, editor, *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP'08)*, volume 5202 of *Lecture Notes in Computer Science*, pages 144–158, Berlin, Heidelberg, 2008. Springer.

[IS10] I. K. Isaev and D. V. Sidorov. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. *Programming and Computer Software*, 36(4):225–236, July 2010.

[ISO12] ISO. *ISO/IEC 14882:2011 Information Technology — Programming Languages — C++*. ISO, February 2012.

[JBH10] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Proceedings* *of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144, Berlin, Heidelberg, 2010. Springer-Verlag.

[JBH11] Matti Järvisalo, Armin Biere, and Marijn Heule. Simulating circuit-level simplifications on CNF. *Journal of Automated Reasoning*, pages 1–37, 2011.

[JBKW08] Jean Christoph Jung, Pedro Barahona, George Katsirelos, and Toby Walsh. Two encodings of DNNF theories. ECAI'08 Workshop on Inference Methods Based on Graphical Structures of Knowledge, July 2008.

[JC09] Himanshu Jain and Edmund M. Clarke. Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts. In *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*, pages 563–568, New York, NY, USA, 2009. ACM.

[JC11] Ajith K. John and Supratik Chakraborty. A quantifier elimination algorithm for linear modular equations and disequations. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 486–503, Berlin, Heidelberg, 2011. Springer.

[JD01] Peer Johannsen and Rolf Drechsler. Formal verification on the RT level computing one-to-one design abstractions by signal width reduction. In *In IFIP International Conference on Very Large Scale Integration (VLSI'01), Montpellier, 2001*, pages 127–132, 2001.

[JLS09] Susmit Jha, Rhishikesh Limaye, and Sanjit Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'2009)*, volume 5643 of *Lecture Notes in Computer Science*, pages 668–674. Springer-Verlag, 2009.

[KFB12] Gergley Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *Proceedings of the* 10th *International Workshop on Satisfiability Modulo Theories (SMT'12)*, pages 44–55, 2012.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KJJP09] Alfred Koelbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. Solver technology for system-level to RTL equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 196–201, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[KKBC12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd Conference on Programming Language Design and Implementation (PLDI 2012)*, pages 193–204. ACM, 2012.

[KP05] Alfred Kölbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, 2005.

[KSJ09] Hyondeuk Kim, Fabio Somenzi, and HoonSang Jin. Efficient term-ITE conversion for satisfiability modulo theories. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2009.

[KVZ09] Johannes Kinder, Helmut Veith, and Florian Zuleger. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proc. of the 10th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, pages 214–228. Springer, 2009.

[LA03] Eric Larson and Todd Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th USENIX Security Symposium*, page 9, Berkeley, CA, USA, 2003. USENIX Association.

[Lar90] Tracy Larrabee. *Efficient Generation of Test Patterns Using Boolean Satisfiability*. PhD thesis, Stanford University, 1990.

[Li00] Chu Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 291–296. AAAI Press, 2000.

[LS10] Rhishikesh Shrikant Limaye and Sanjit A. Seshia. Beaver: An SMT solver for quantifier-free bit-vector logic. Technical Report UCB/EECS-2010-67, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2010.

[MCB06] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 532–535, New York, NY, USA, 2006. ACM.

[Min92] Shin-ichi Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Proceedings of the Synthesis and Simulation Meeting and International Interchange (SASIMI'92)"*, 1992.

[Min96] Shin-ichi Minato. Generation of BDDs from hardware algorithm descriptions. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, pages 644–649, Washington, DC, USA, 1996. IEEE Computer Society.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[MPFH11] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In Eran Yahav, editor, *Static Analysis: Proceedings of the 18th International Symposium*, volume 6887 of *Lecture Notes in Computer Science*, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[MSV06] Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon. Automatic memory reductions for RTL model verification. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '06, pages 786–793, New York, NY, USA, 2006. ACM.

[MV12] Laurant D. Michel and Pascal Van Hentenryck. Constraint satisfaction over bit-vectors. In M. Milano, editor, *Constraint Programming: Proceedings of the 2012 Conference*, volume 7514 of *Lecture Notes in Computer Science*, pages 527–543. Springer, 2012.

[NLLC06] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. BIRD: Binary interpretation using runtime disassembly. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370, Washington, DC, USA, 2006. IEEE Computer Society.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Proceedings of the 16th International Conference on Rewriting Techniques and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

[NRJ$^+$07] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *AI Magazine*, 28(3):13–30, 2007.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100, New York, NY, USA, 2007. ACM.

[NSSS12] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code.

In R. Jhala and A. Igarashi, editors, *APLAS 2012: Proceedings of the 10th Asian Symposium on Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2012.

[OSC09] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, September 2009.

[PG86] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[RD06] John Regehr and Usit Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tool Support for Embedded Systems*, pages 34–43, New York, NY, USA, 2006. ACM.

[RR04] John Regehr and Alastair Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–143, New York, NY, USA, 2004. ACM.

[RSY04] Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In B. Steffen and G. Levi, editors, *Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2004.

[Rud86] Richard L. Rudell. Multiple-valued logic minimization for PLA synthesis. Technical Report UCB/ERL M86/65, EECS Department, University of California, Berkeley, 1986.

[Rus99] David M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, January 1999.

[SBDL01] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *Proceed-*

ings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01), pages 29–37, Washington, DC, USA, 2001. IEEE Computer Society.

[SD11] Sol Swords and Jared Davis. Bit-blasting ACL2 theorems. In David Hardin and Julien Schmaltz, editors, Proceedings 10th International Workshop *on the ACL2 Theorem Prover and Its Applications,* Austin, Texas, USA, November 3-4, 2011, volume 70 of *Electronic Proceedings in Theoretical Computer Science*, pages 84–102. Open Publishing Association, 2011.

[Seb07] Roberto Sebastiani. Lazy satisability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(3-4):141–224, 2007.

[Sed98] Robert Sedgewick. *Algorithms in C.* Addison-Wesley, third edition, 1998.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[Smi11] Eric Whitman Smith. *Axe, an automated formal equivalence checking tool for programs.* PhD thesis, Stanford University, 2011.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257, Berlin, Heidelberg, 2009. Springer-Verlag.

[Tse83] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning, Vol. 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, 1983. Originally published as "O slozhnosti vyvoda v ischislenii vyskazyvaniy", *Zapiski Nauchnykh Seminarov LOMI* **8**:234-259, Steklov Inst. Math., Leningrad, 1968.

[War02] Henry S. Warren Jr. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Weg60] Peter Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5):322, May 1960.

[WHdM10] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. Efficiently solving quantified bit-vector formulas. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 239–246, Austin, TX, 2010. FMCAD Inc.

[XA05] Yichen Xie and Alex Aiken. Scalable error detection using Boolean satisfiability. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, New York, NY, USA, 2005. ACM.

[XHHLB08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32(1):565–606, 2008.

[ZKC01] Zhihong Zeng, Priyank Kalla, and Maciej Ciesielski. LPSAT: A unified approach to RTL satisfiability. In *DATE 2001*, pages 398–402. IEEE Press, 2001.