Bit-Vector Rewriting with Automatic Rule Generation

Alexander Nadel

Intel Corporation, P.O. Box 1659, Haifa 31015, Israel
`alexander.nadel@intel.com`

Abstract. Rewriting is essential for efficient bit-vector SMT solving. The rewriting algorithm commonly used by modern SMT solvers iteratively applies a set of ad hoc rewriting rules hard-coded into the solver to simplify the given formula at the preprocessing stage. This paper proposes an automatic approach to rewriting. The solver starts each invocation with an empty set of rewriting rules. The set is extended by applying at run-time an automatic SAT-based algorithm for new rewriting rule generation. The set of rules differs from instance to instance. We implemented our approach in the framework of an algorithm for equivalence and constant propagation, called 0-saturation, which we extended from purely propositional reasoning to bit-vector reasoning. Our approach results in a substantial performance improvement in a state-of-the-art SMT solver over various SMT-LIB families.

1 Introduction

Bit-vector reasoning is applied in a variety of domains [10, 8, 13, 21, 15, 18, 14]. Modern bit-vector solvers, such as Boolector [6] and Mathsat [7], employ *rewriting* [11, 1, 9] at the preprocessing stage. Rewriting applies a set of rewriting rules until fixed point to simplify the formula. For example, the statement $x = \texttt{bvule}(0, z)$ (where `bvule` stands for unsigned-less-than-equal) can be rewritten to $x = \mathit{true}$, while $x = \texttt{bvadd}(y, -y)$ (where `bvadd` stands for addition) can be rewritten to $x = 0$. Normally, the rewriting rules are designed manually by the developers of each solver and are embedded into the solver's code offline (that is, during solver development time). The same set of rules is used irrespective of the input instance. Rewriting is applied to simplify the directed acyclic graph (DAG) representing the formula. The number of rewriting rules can reach into the hundreds. For example, citing [9], "in MathSAT, close to 300 rewrite rules have been defined".

We propose an alternative approach to rewriting. Table 1 summarizes the differences between our approach and the standard method. In our approach, the solver starts with an empty set of rules. It then attempts to generate a rule on the fly at run-time whenever it identifies a situation where a rewriting rule is likely to be applied. For example, when one of the operands of an operation belongs to a set of pre-defined constants, such as 0, as in $x = \texttt{bvule}(0, z)$, the solver checks whether the result can be rewritten to a pre-defined constant. To that end, it uses incremental SAT solver invocations over a new CNF instance comprising the formula under consideration ($x = \texttt{bvule}(0, z)$ in our case), bit-blasted to CNF. In our case, the algorithm will learn that $x = \texttt{bvule}(0, z)$ can be rewritten to $x = true$. The process of trying to generate a rule is also triggered when two operands are related by a simple function, such as unary minus, as in $x = \texttt{bvadd}(y, -y)$. Hence, the algorithm will realize that $x = \texttt{bvadd}(y, -y)$ can be rewritten to $x = 0$. Whenever a new rule is generated, it is immediately applied and also stored in a hash table of rules, which is reused as long as the solver is alive. Hence the set of rules is instance-specific.

Our algorithm is implemented in the framework of 0-saturation [19, 12, 17], a process of constant and equivalence propagation, initially proposed in the context of propositional reasoning. We extended 0-saturation to handle bit-vector reasoning. The added value of 0-saturation over DAG rewriting is that 0-saturation enables propagating equivalences implied by user-given assertions.

We implemented our algorithm in Intel's SMT solver Hazel on top of an existing preprocessor. Section 4 shows that using our approach pays off experimetally. It improves the performance of Hazel on 20 out of 23 tested SMT-LIB [4] families. New Hazel outperforms the leading SMT solvers Boolector and Mathsat on most of the tested families. Moreover, there are 10 families on which new Hazel outperforms base Hazel, Boolector and Mathsat significantly: it either solves more instances or is at least 2x faster. The overhead of generating the rules at run-time rather than offline is negligible in practice.

The most relevant previous work is [20], where an offline instance-generic *automatic* generator of rewriting rules for bit-vector reasoning is proposed. Unfortunately, it does not work in practice [20]. The algorithm was halted after generating approximately 120,000 rules, and the generated rules could not simplify real-world problems. In contrast, our approach explores a narrower rule space and, being instance-specific, restricts it further by exploring only rules relevant to the input problem. [20] successfully applies a *semi-automatic* algorithm. First the algorithm automatically generates rule candidates for a specific width range from a rule space restricted to equivalences. Then, based on unspecified criteria, a human manually chooses which of those rules to embed into the solver. One problem that might have prevented full automation of this algorithm is that the set of candidates is width-specific, hence manual effort is required to choose width-independent rules only. Our approach avoids this problem since, being instance-specific, it knows the widths of the operations it needs to generate rules for. Offline instance-generic automatic rule generation is also applied for peephole super-optimization [2, 3] and symbolic binary execution [18].

In what follows, Section 2 provides preliminaries and describes 0-saturation. Section 3 introduces bit-vector 0-saturation with automatic rule generation. Section 4 provides experimental results. We conclude in Section 5.

	Standard Method	Our Approach
When are the rules created?	Offline (solver development time)	Run-time
How are the rules generated?	Manually	Automatically
Where are the rules stored?	Hard-coded	Hash table
Are the rules instance-specific?	No	Yes
Rewriting framework	DAG-based rewriting	0-Saturation

Table 1. Comparing Our Approach to Rewriting to the Standard Method.

2 Preliminaries and 0-Saturation

We start with some basic notions. A *bit* is a Boolean variable, which can be interpreted as 0 or 1. A *bit-vector* v of width n is a sequence of n bits, where the right-most bit is the least significant bit. The set of all bit-vectors of width n is denoted by $\mathcal{BV}_n$. A *constant* is a bit-vector whose every bit is substituted by 0 or 1. The set of all constants for width n is denoted by $\mathcal{BC}_n$. We do not define a separate Boolean type `Bool` for propositional variables, but use $\mathcal{BV}_1$ to formally represent propositional variables (unlike in the SMT-LIB 2.0 language [5]).

We denote bitwise negation by $\sim$. Generalizing propositional logic, we let a bit-vector *literal* be a bit-vector variable v or its bitwise negation $\sim v$. We denote the set of bit-vector literals of width n by $\mathcal{BL}_n$. Next we define a binary operation and a triplet. Our definition of a triplet is a strict generalization of the definition of a propositional triplet [19, 12, 17].

Definition 1 (Binary Operation; Binary Operation Width; Predicate/Non-Predicate Binary Operation). *A* binary operation o *is a function that maps any two constants in* $\mathcal{BC}_n$ *to a constant in* $\mathcal{BC}_m$*, where* $w(o) = n$ *is the* operation width. *Each binary operation belongs to one of the following categories:*

1. *A* predicate operation *has* $m = 1$ *and* $w(o) > 1$.
2. *A* non-predicate operation *has* $m = w(o)$.

Definition 2 (Triplet; Triplet Member/Width/Type; Predicate/Non-Predicate Triplet). *A* triplet *is an application of a binary operation* o*:* $x = o(y, z)$*, where* $y, z \in \mathcal{BL}_n \cup \mathcal{BC}_n$*,* $x \in \mathcal{BL}_m \cup \mathcal{BC}_m$*,* $w(t) = w(o) = n$ *is the* triplet width*, and* x, y, z *are* triplet members. *The pair* $\{o, w(o)\}$ *constitutes the* triplet type. *A triplet is* predicate/non-predicate *iff* o *is predicate/non-predicate, respectively.*

For example, given $y, z \in \mathcal{BV}_{n>1}$, $x = \texttt{bvule}(y, z)$ is a predicate triplet, while $x = \texttt{bvadd}(y, z)$ is a non-predicate triplet. We define a binding as follows:

Definition 3 (Binding). *A* binding $x = y$ *for* $x, y \in \mathcal{BL}_n \cup \mathcal{BC}_n$ *stands for the equality between* x *and* y.

Next we review the *0-saturation* algorithm [19, 12, 17], initially presented as a way to simplify a propositional formula by constant and equivalence propagation. Our paper generalizes 0-saturation to handle bit-vectors. We provide a generic framework for the algorithm that fits both the propositional and the bit-vector cases.

Consider the 0-saturation algorithm in Alg. 1. Given a set of triplets T and a set of bindings B, 0-saturation carries out *in-place* rewriting of T to a set of triplets equisatisfiable to $T \wedge B$, but potentially having fewer triplets and variables than T, since the algorithm may render triplets tautological and replace variables by the representatives of their respective equivalence classes (see below). The algorithm classifies the triplets into three categories by associating one of the following statuses with each triplet:

1. *unknown*: The triplet must be evaluated by the main loop of the algorithm (lines 5 to 7). Initially all the triplets are unknown (see line 2).
2. *tautological*: The triplet is a tautology. Once a triplet becomes tautological, its status will never change. Tautological triplets are essentially removed from T.
3. *active*: No further information can be learned from the triplet at this stage. An active triplet becomes unknown if and when one of its members is changed.

The algorithm divides the set of variables and constants into separate equivalence classes, where one literal of each variable appears in the equivalence classes. Each equivalence class has one and only one *representative*. Initially each class contains one constant or one variable serving as the representative (see line 3). For example, for the propositional case, assuming that both constants 0 and 1 appear, the initial equivalence classes would look as follows: $[\underline{0}], [\underline{1}], [\underline{v_1}], [\underline{v_2}], \ldots, [\underline{v_n}]$ (representatives are underlined). Line 4 *merges* each binding $\{p = q\} \in B$, that is, it merges the equivalence classes of p and q by applying the function MERGE.

Consider the function MERGE at line 10. If a literal or a constant is merged with its negation or if two different constants are merged, a contradiction exception is thrown. Otherwise, MERGE negates all the members of the equivalence classes of p and/or q, if p and/or q appear negated in their respective equivalence classes; then, it merges the equivalence classes of p and q. The new representative can be picked in an arbitrary manner, except that a constant must always be a representative. Line 14 forces the algorithm to reevaluate each triplet containing the former representative q or its negation by changing the status of such triplets to unknown. Finally, line 15 replaces all the occurences of the former representative q or its negation $\sim q$ with the new representative p or its negation $\sim p$, respectively.

The main loop of the algorithm (line 5) tries to rewrite the formula by applying the function PROPAGATE over unknown triplets. PROPAGATE may change the status of t and other triplets. The main loop operates until no more unknown triplets exist or until MERGE discovers a contradiction. We assume that a contradiction in MERGE is handled through the exception mechanism.

The function PROPAGATE (line 16) comprises the heart of the algorithm. It tries, if possible, to apply rewriting rules in order to infer and merge new bindings and render the triplet under evaluation tautological. We discuss the core rewriting algorithm at length in Section 3.

3 Bit-Vector 0-Saturation with Automatic Rule Generation

This section introduces our algorithm for automatic rule generation in the context of bit-vector 0-saturation. Section 3.1 provides the algorithm's high-level scheme, while Section 3.2 refines and formalizes the algorithm.

In order to look for rewriting rules systematically, it is necessary to define all the possible conditions that may trigger a rule and all the possible conclusions

Algorithm 1 0-Saturation Algorithm.

```
1: function 0-SATURATION(Triplets $S$, Bindings $B$)
2:     For each $t \in S$, set $stt(t) := unknown$
3:     Initialize equivalence classes to hold one variable and one constant each
4:     For each $\{p = q\} \in B$, MERGE($p$, $q$)
5:     while There exists $t \in S$, such that $stt(t) = unknown$ do
6:         $t := t \in S$, such that $stt(t) = unknown$
7:         $stt(t) :=$ PROPAGATE($t$)

8: function NEGATEEQCLASSIFREQUIRED(Literal or constant $r$)
9:     if ($r$ is negated in its eq. class) then Negate all the members of $r$'s eq. class
10: function MERGE(Literal or constant $p$, Literal or constant $q$)
11:     if ($p = \sim q$) or (($p \neq q$) and $p$ and $q$ are constants) then throw contradiction
12:     NEGATEEQCLASSIFREQUIRED($p$); NEGATEEQCLASSIFREQUIRED($q$)
13:     Merge the eq. classes $[\underline{p}, v_1^p, \dots, v_k^p]$ and $[\underline{q}, v_1^q, \dots, v_l^q]$ into
        $[\underline{p}, v_1^p, \dots, v_k^p, q, v_1^q, \dots, v_l^q]$   ▷ See text for the way to pick the new representative
14:     For any triplet $t$ containing $q$ or $\sim q$, s.t. $stt(t) = active$, set $stt(t) := unknown$
15:     Replace $q$ and $\sim q$ by $p$ and $\sim p$, respectively, in any triplet containing $q$ or $\sim q$
16: function PROPAGATE(Triplet $t$)
17:     if A rewriting rule is applicable for $t$ then
18:         Use MERGE to merge new bindings resulting from applying the rule
19:         if $t$ becomes a tautology after applying the rule return $tautological$
20:     return $active$
```

that a rule may imply. One can then learn the actual rules implied by the logic under consideration.

For the propositional case, an exhaustive list of rewriting rules for 0-saturation of the following form per a Boolean operation is provided in [17]: a rule is triggered when one of the members of a triplet t is either the constant 0 or 1 or is equivalent up to negation to another member. The rule implies either a contradiction or one or two new bindings of t members or their negations to the constants or to other t members. Whenever a rule is triggered, t becomes tautological. For example, one of the rewriting rules listed in [17] would infer the binding $\sim x = y$ from the triplet $\sim x = \texttt{and}(y, y)$ (where `and` stands for the Boolean and), rendering the triplet tautological. In [17] the rules are hard-coded into the solver, and no formal procedure for generating the rules is provided.

3.1 Automatic Rule Generation: High-Level Algorithm

In our approach to bit-vector 0-saturation, a rule may be triggered when, given a triplet $x = o(y, z)$, one of its members x, y, z is a pre-defined *rewriting constant* and/or when one member constitutes a pre-defined *rewriting function* of another member (rewriting constants and functions will be defined shortly). Usually, applying a rewriting rule binds triplet members to rewriting constants or to rewriting functions of other members.

Next we define a successor, a predecessor and a neighbor. The two equations in the following definition result from substituting x by $\sim x$ and $-x$, respectively, in the definition of unary minus $-x = \sim x + 1$.

Definition 4 (Successor; Predecessor; Neighbor). *The* successor *of $x \in \mathcal{BL}_n \cup \mathcal{BC}_n$ is $x + 1 = -\sim x$; the* predecessor *of x is $x - 1 = \sim -x$. x's successor or predecessor is x's* neighbor.

The following definitions capture the notion of rewriting constants, rewriting functions, rewriting identities, and rewriting values:

Definition 5 (Rewriting Constant). *The set of* Rewriting Constants $\mathcal{RC}$ *comprises the following constants defined for every type $\mathcal{BC}_w$:*

1. $0 = \underbrace{0 \ldots 0}_{w}$
2. $-1 = \underbrace{1 \ldots 1}_{w}$
3. $1 = \underbrace{0 \ldots 0}_{w-1}1$
4. $2 = \underbrace{0 \ldots 0}_{w-2}10$ *for* $w \geq 2$; $2 = 0$ *for* $w = 1$
5. $-2 = \underbrace{1 \ldots 1}_{w-1}0$

Definition 6 (Rewriting Function). *Given a bit-vector formula e, the set of* Rewriting Functions $\mathcal{RF}_e$ *comprises the following parametrized set of functions:*

1. $f_1(e) = e$
2. $f_2(e) = e - 1 = \sim -e$
3. $f_3(e) = e - 2 = \sim -\sim -e$
4. $f_4(e) = e + 1 = -\sim e$
5. $f_5(e) = e + 2 = -\sim -\sim e$
6. $f_6(e) = -e$
7. $f_7(e) = -e - 1 = \sim e$
8. $f_8(e) = -e - 2 = \sim -\sim e$
9. $f_9(e) = -e + 1 = -\sim -e$
10. $f_{10}(e) = -e + 2 = -\sim -\sim -e$

Definition 7 (Rewriting Identity; x-identity; y-identity; z-identity). *The set of* rewriting identities $I = \{i_x, i_y, i_z\}$ *comprises the* x-identity i_x, *the* y-identity i_y, *and the* z-identity i_z.

The rewriting identities i_x, i_y, and i_z are used to represent situations where, given a triplet $x = o(y, z)$, x, y, and z, respectively, are neither rewriting constants nor rewriting functions of another triplet member.

Definition 8 (Rewriting Value; Trivial Rewriting Value). *Let $x = o(y, z)$ be a triplet. The set $I \cup \mathcal{RC} \cup \mathcal{RF}_x \cup \mathcal{RF}_y \cup \mathcal{RF}_z$ comprises all the* rewriting values. *The rewriting value is* trivial *if and only if it is a rewriting identity.*

If one or more triplet members is a non-trivial rewriting value (that is, it is either a rewriting constant or a rewriting function of another triplet member), a rewriting rule may be triggered. Triplet members may be bound to non-trivial rewriting values as a result of applying a rule. We picked the rewriting values bearing the following reasons in mind:

1. The constants 0, -1 and 1 are essential to be able to generate a variety of rules for multiple operations. For example, both $x = \texttt{bvadd}(y, 0)$ and $x = \texttt{bvmul}(y, 1)$ (where `bvmul` stands for multiplication) can be rewritten to $x = y$, while $x = \texttt{bvule}(y, -1)$ can be rewritten to $x = -1$.
2. Bitwise negation is essential for rewriting bitwise operations. For example, $-1 = \texttt{bvand}(y, {\sim}y)$ is a contradiction (where `bvand` stands for bitwise and). In addition, having bitwise negation and the constants -1,0 ensures that our procedure covers the propositional case.
3. Unary minus is essential for rewriting arithmetic operations. For example, $x = \texttt{bvadd}(y, -y)$ can be rewritten to $x = 0$.
4. Capturing the neighbors of a literal and the neighbors of its unary minus is useful for rewriting a variety of operations. For example, $x = \texttt{bvadd}(y, -y-1)$ can be rewritten to $x = -1$. We have chosen to look at neighbors up to depth 2 in order to attempt to create rules to capture sequences of successors and predecessors. To that end, we have also included the constants 2 and -2.
5. Our algorithm can check whether triplet members are rewriting values instantaneously by holding a pointer to each literal's negation and unary minus, since all our rewriting values can be expressed using constants, bitwise negation and unary minus only.

Next we define the notions of a premise and a skeleton, where the expression $g^{e_1 \mapsto e_2}$ stands for g, where each instance of e_1 is *substituted* by e_2.

Definition 9 (Premise). *Given a formula F, a triplet $t \equiv x = o(y, z)$, the ordered set $\{\sigma_x, \sigma_y, \sigma_z\}$, where $\sigma_x \in \{i_x\} \cup \mathcal{RC} \cup \mathcal{RF}_y \cup \mathcal{RF}_z$, $\sigma_y \in \{i_y\} \cup \mathcal{RC} \cup \mathcal{RF}_x \cup \mathcal{RF}_z$, and $\sigma_z \in \{i_z\} \cup \mathcal{RC} \cup \mathcal{RF}_x \cup \mathcal{RF}_y$, is a* premise *if the following conditions hold*

1. *$F \implies x = \sigma_x^{i_x \mapsto x} \wedge y = \sigma_y^{i_y \mapsto y} \wedge z = \sigma_z^{i_z \mapsto z}$*
2. *One of the values $\{\sigma_x, \sigma_y, \sigma_z\}$ is non-trivial.*

Definition 10 (Skeleton; Skeleton Member). *Given a triplet $t \equiv x = o(y, z)$ and a premise $\{\sigma_x, \sigma_y, \sigma_z\}$, the formula $S \equiv \sigma_x^{i_x \mapsto x} = o(\sigma_y^{i_y \mapsto y}, \sigma_z^{i_z \mapsto z})$ is a* skeleton *and $\left\{\sigma_x^{i_x \mapsto x}, \sigma_y^{i_y \mapsto y}, \sigma_z^{i_z \mapsto z}\right\}$ are* skeleton members.

In the definitions above, we assume that F is the current formula (that is, a set of triplets, and, possibly, bindings), maintained by the 0-saturation algorithm. If F implies that at least one of the members of a given triplet $t \equiv x = o(y, z)$ is a non-trivial rewriting value, a premise is well-defined. For example, given the triplet $x = \texttt{bvadd}(y, 1)$ and assuming $x = y + 1$, the sets

$\{i_x, i_y, 1\}$, $\{y+1, i_y, i_z\}$, and $\{y+1, i_y, 1\}$ are all premises. Premise definition will be refined in Section 3.2.

A skeleton is used to isolate an application of the triplet's operation, given a premise, irrespectively of irrelevant triplet members. In our example, $x = \texttt{bvadd}(y, 1)$ is the skeleton given the premise $\{i_x, i_y, 1\}$; $y + 1 = \texttt{bvadd}(y, z)$ is the skeleton given the premise $\{y+1, i_y, i_z\}$; and $y + 1 = \texttt{bvadd}(y, 1)$ is the skeleton given the premise $\{y+1, i_y, 1\}$.

Let us move now to a high-level sketch of our algorithm.

1. When the SMT solver starts, it applies 0-saturation as part of preprocessing. The solver maintains a hash table of rewriting rules for each triplet type that had at least one rule generated for it. The hash tables map each rule's premise to the rule's *conclusion*, which can either be a contradiction, a set of bindings of triplet members to rewriting values, or empty (meaning that no action can be carried out under the current premise). The conclusion also defines whether the triplet becomes a tautology after application of the rule.
2. Whenever an unknown triplet t is evaluated by the propagation algorithm Propagate (line 16), the solver first checks whether both y and z are constants of any kind (not necessarily rewriting constants). If they are, it merges x with the constant $o(y, z)$, marks the triplet tautological, and exits. Otherwise, it carries out *premise detection*, that is, it checks whether the current formula implies any premise, given t.[1] If a premise is not detected, no further information can be learned from the triplet and the algorithm renders it active.
3. If a premise is detected, the solver checks if a rule with the corresponding premise appears in the hash table. If it does, the solver *applies* the rule in the following sense. If the conclusion is empty, the solver renders the triplet active (storing empty conclusions prevents regeneration of the same empty conclusion over and over again). If the conclusion is a contradiction, a contradiction exception is thrown. If the conclusion is a set of bindings, the solver binds triplet members to their corresponding values. The solver may also mark the triplet tautological, if required by the conclusion.
4. If a premise is detected, but no corresponding rule is found in the hash table, the solver enters the *conclusion generation* stage. It bit-blasts the skeleton, corresponding to the triplet and the premise, to a fresh SAT instance. Then, the solver checks for a contradiction in the instance and for all the possible new bindings between the skeleton members and non-trivial rewriting constants or rewriting functions of other skeleton members using incremental SAT invocations. If there is no contradiction and no bindings can be learned, the conclusion is empty. The newly generated conclusion is either a contradiction, an empty set, or a set of bindings. In the latter case, the conclusion also specifies whether the triplet becomes tautological (more details are provided in Section 3.2). In the end, the conclusion is inserted into the hash table, and the new rule is applied as described in the previous step.

[1] Let us assume for now that an arbitrary premise is picked whenever more than one premise exists. Section 3.2 discusses premise redundancy and premise subsumption.

We distinguish between tautological and non-tautological conclusions, since, unlike in propositional case, in bit-vector 0-saturation, it is possible to conceive of an operation and a rule, where applying the rule would not make the corresponding triplet a tautology. For example, one could design an operation such that when x becomes 0, y must also become 0, but while z's range is reduced, z is still neither fixed nor a don't care. We did not find such operations in the current SMT-LIB language, but our algorithm takes into consideration their possible existence.

Note that we restrict ourselves to rewriting rules of a certain pre-defined format, which is most relevant to 0-saturation. It is possible to extend our procedure, e.g., by rewriting triplets to triplets of a different type or by considering more than one triplet at once for rewriting. We leave the exploration of such possibilities to future work.

Our procedure can be applied in a straightforward manner given a bit-vector formula in the SMT-LIB 2.0 language [5], since the vast majority of that language's operations are binary, and hence can be represented as triplets. However, some of the operations, such as unary minus, are unary, while `ite` (if-then-else) has 3 operators. In principle our procedure can easily be extended to accomodate non-binary operators. However, in our current implementation we use simple hard-coded rules for unary operators and `ite`.

3.2 Refining and Formalizing the Algorithm

This section refines the algorithm described in the previous section, bearing the following three main goals in mind:

1. *Eliminating premise redundancy* mainly by disallowing syntactically different but semantically identical premises so as to prevent the algorithm from generating essentially the same rule multiple times. For example, only the first of the two sets $\{y, i_y, i_z\}$, $\{i_x, x, i_z\}$ will qualify as a premise by our refined definition, given the triplet $v = \texttt{bvadd}(v, z)$.
2. *Generating stronger conclusions* by binding triplet members to rewriting constants rather than to rewriting functions, whenever possible. For example, given the triplet $x = \texttt{bvand}(y, z)$ and the premise $\{-1, i_y, i_z\}$, binding both y and z to -1 is likely to simplify the instance more substantially than just binding y to z.
3. *Generating weaker premises* by disqualifying rules whose premise would be subsumed by another rule with the same conclusion. For example, if for some triplet t the same conclusion is implied by both $p = \{i_x, -z+2, i_z\}$ and $p' = \{-1, -z+2, i_z\}$, then p is the desirable premise for the new rule, since it leaves more opportunities for applying the rule by not restricting x.

We refine the notion of rewriting values by defining the notions of x-value, y-value, and z-value, the rewriting values for x, y, and z, respectively.

Definition 11 (x-value). *Given a triplet $x = o(y, z)$, the rewriting value σ_x is an* x-value *if it belongs to the set $V_x \cup \{i_x\}$, where V_x is defined as follows:*

1. *$V_x := \{0, -1\}$ if o is a predicate operation*
2. *$V_x := \mathcal{RC} \cup \mathcal{RF}_y \cup \mathcal{RF}_z$ if o is a non-predicate operation and $w > 1$*
3. *$V_x := \{0, -1, y, {\sim}y, z, {\sim}z\}$ if o is a non-predicate operation and $w = 1$*

For a predicate operation, a non-trivial x-value can only be one of the constants $\{0, -1\}$. For non-predicate operations, a non-trivial x-value can belong to the set $\mathcal{RC} \cup \mathcal{RF}_y \cup \mathcal{RF}_z$ as expected, except in cases where the width is 1, and the set is refined to eliminate redundancies.

Definition 12 (y-value). *Given a triplet $x = o(y, z)$, the rewriting value σ_y is a* y-value *if it belongs to the set $V_y = \mathcal{RC} \cup \mathcal{RF}_z \cup \{i_y\}$.*

Definition 13 (z-value). *Given a triplet $x = o(y, z)$, the rewriting value σ_z is a* z-value *if it belongs to the set $V_z = \mathcal{RC} \cup \{i_z\}$.*

A y-value or a z-value may not belong to $\mathcal{RF}_x$ to eliminate redundancy. This is because any expression of the form $y = f(x) \in \mathcal{RF}_x$ or $z = f(x) \in \mathcal{RF}_x$ can be represented as $x = f^{-1}(y) \in \mathcal{RF}_y$ or $x = f^{-1}(z) \in \mathcal{RF}_z$, respectively. For the same reason, a z-value may not belong to $\mathcal{RF}_y$ (hence, a non-trivial z-value may only be a rewriting constant). Note that the set $\mathcal{RF}_x$ is redundant and is used no more. Now we can formulate a refined notion of a premise.

Definition 14 (Premise (Refined)). *Given a formula F, a triplet t, the ordered set $\{\sigma_x, \sigma_y, \sigma_z\}$, where $\sigma_x \in V_x$, $\sigma_y \in V_y$, $\sigma_z \in V_z$, is a* premise *if $F \implies x = \sigma_x^{i_x \mapsto x} \wedge y = \sigma_y^{i_y \mapsto y} \wedge z = \sigma_z^{i_z \mapsto z}$ and* one *of the following conditions hold:*

1. *One and only one of the values $\{\sigma_x, \sigma_y, \sigma_z\}$ is non-trivial, or*
2. *$\sigma_x \in \mathcal{RC}$ and $\sigma_y \in \mathcal{RC} \cup \mathcal{RF}_z$ and $\sigma_z = i_z$, or*
3. *$\sigma_x \in \mathcal{RF}_y$ and $\sigma_y = i_y$ and $\sigma_z \in \mathcal{RC}$, or*
4. *$\sigma_x \in \mathcal{RF}_z$ and $\sigma_y \in \mathcal{RC}$ and $\sigma_z = i_z$, or*
5. *$\sigma_x \in \mathcal{RF}_y$ and $\sigma_y \in \mathcal{RF}_z$ and $\sigma_z = i_z$*

Our refined definition of a premise is aimed towards eliminating redundancy. In particular, the case where both y and z are constants is *not* part of the definition, since it is covered by the higher-level algorithm, which simply binds x to the constant $o(y, z)$. We also skip all the cases where some triplet member v is a constant and there is another triplet member u, such that $u = f(v)$ (for example, $\{y + 1, 1, i_z\}$ is thus skipped). This is because such cases are mostly covered by the case where both v, u are constants ($\{2, 1, i_z\}$ in our case), while cases which are not covered by our rewriting constant set are simply skipped (e.g., $\{y + 2, 2, i_z\}$ would be covered by $\{4, 2, i_z\}$, but 4 is not a rewriting constant). In addition, we skip the case where both x and y are functions of z, since it is mostly covered by the case where x is a function of y and y is a function of z.

Next we define a total order between values, which is essential for generating stronger conclusions, and formalize the notion of a conclusion.

Definition 15 (Order over Values). *The following order relation induces a total order between any two values in $I \cup \mathcal{RF}_y \cup \mathcal{RF}_z \cup \mathcal{RC}$: $I < \mathcal{RF}_y \cup \mathcal{RF}_z < \mathcal{RC}$.*

Definition 16 (Conclusion; Contradictory/Empty/Partial/Tautological/Interesting Conclusion). *Given a triplet t and a premise $\{\sigma_x, \sigma_y, \sigma_z\}$, c is a* conclusion *if the following conditions hold. Let $S \equiv \sigma_x^{i_x \mapsto x} = o(\sigma_y^{i_y \mapsto y}, \sigma_z^{i_z \mapsto z})$ be the skeleton.*

1. *If $S \implies \bot$, then $c \equiv \bot$, in which case the conclusion is* contradictory*.*
2. *If $S \not\Longrightarrow \bot$, then $c \equiv \{\rho_x, \rho_y, \rho_z, taut \in \{false, true\}\}$, where $\rho_x \in V_x$, $\rho_y \in V_y$, $\rho_z \in V_z$, and the following equations hold:*
   - *(a) ρ_z is the maximal value, such that $S \implies \sigma_z^{i_z \mapsto z} = \rho_z^{i_z \mapsto z}$*
   - *(b) ρ_y is the maximal value, such that $S \implies \sigma_y^{i_y \mapsto y} = \rho_y^{i_y \mapsto y}$*
   - *(c) ρ_x is the maximal value, such that $S \implies \sigma_x^{i_x \mapsto x} = \rho_x^{i_x \mapsto x}$, where the order is refined in the following two cases: if $\sigma_x \in \mathcal{RF}_y$, then $I < \mathcal{RF}_y < \mathcal{RF}_z < \mathcal{RC}$; if $\sigma_x \in \mathcal{RF}_z$, then $I < \mathcal{RF}_z < \mathcal{RF}_y < \mathcal{RC}$*
   - *(d) $taut = true$ iff $\sigma_x^{i_x \mapsto x} = \rho_x^{i_x \mapsto x} \wedge \sigma_y^{i_y \mapsto y} = \rho_y^{i_y \mapsto y} \wedge \sigma_z^{i_z \mapsto z} = \rho_z^{i_z \mapsto z} \implies S$*

If the conclusion is $\{\sigma_x, \sigma_y, \sigma_z, false\}$, it is empty*. The conclusion is* interesting *if it is not empty. An interesting conclusion with $taut = false$ or $taut = true$ is* partial *or* tautological*, respectively.*

A conclusion can be characterized as follows:

1. If the skeleton is contradictory, then the conclusion must be contradictory.
2. A conclusion always exists, since setting each value in the conclusion to the corresponding value in the premise comprises the empty conclusion, if no other conclusion is available.
3. Any non-empty conclusion (if available) is preferred to the empty conclusion.
4. Rewriting constants are always preferred to rewriting functions.
5. A non-contradictory conclusion is tautologial iff binding its values to the premise values implies the skeleton.

Finally, we formally define a rule aiming towards generating rules with weaker premises as we discussed.

Definition 17 (Rule; Contradictory/Empty/Partial/Tautological/Interesting Rule). *Given a triplet t, a premise $p = \{\sigma_x, \sigma_y, \sigma_z\}$, and a conclusion c, $r \equiv p \implies c$ is a* rule *if there exists no premise $p' = \left\{\sigma'_x, \sigma'_y, \sigma'_z\right\}$ that* subsumes *p, where p' subsumes p iff:*

1. *c is a conclusion, given p', and*
2. *$\sigma'_x \in \{\sigma_x, i_x\}$ and $\sigma'_y \in \{\sigma_y, i_y\}$ and $\sigma'_z \in \{\sigma_z, i_z\}$, and*
3. *($\sigma'_x = i_x$ and $\sigma_x \neq i_x$) or ($\sigma'_y = i_y$ and $\sigma_y \neq i_y$) or ($\sigma'_z = i_z$ and $\sigma_z \neq i_z$)*

The rule is contradictory/empty/partial/tautological/interesting *if c is contradictory/empty/partial/tautological/interesting, respectively.*

We are now ready to present our algorithm for bit-vector 0-saturation with automatic rule generation. Consider Alg. 2. The main function PropagateBV is designed so as to be inserted into Alg. 1 in place of Propagate. PropagateBV receives a triplet t. It may apply empty rewriting rules and/or zero or one interesting rules for the triplet. It returns the status, that is, either *unknown*, *active*, or *tautological*. The status is active if no rules or only empty rules could be applied; it is tautological if a tautological rule was applied, and it is unknown if a partial rule was applied. The function may also throw a contradiction exception. Any expression of the form "**if** c **then** s_1 & s_2" in the pseudo-code of Alg. 2 means that if c holds, the algorithm applies s_1 and then immediately applies s_2.

First, the algorithm enters the premise detection stage to check whether the triplet has a premise. The structure and the order of premise detection (lines 3 to 14) ensure that any generated rule will be correct w.r.t both premise subsumption as dictated by Def. 17 and premise redundancy as dictated by Def. 14. PROPAGATEBV triggers the process of conclusion generation by invoking the function GETC for any identified premise. GETC returns active for an empty conclusion, tautological for a tautological conclusion, and unknown for a partial conclusion. We will describe GETC later.

When an empty rule is returned by GETC, PROPAGATEBV will continue looking for other rules. PROPAGATEBV returns immediately if it finds a non-empty rule. Consider a situation where different non-empty rules may be applied for a given triplet. For example, both the rules $\{i_x, -1, i_z\} \implies \{-1, -1, -1, \text{true}\}$ and $\{i_x, i_y, -1\} \implies \{-1, -1, -1, \text{true}\}$ are applicable for rewriting the triplet $x = \texttt{bvor}(-1, -1)$ (where `bvor` stands for bitwise or). Our algorithm will generate one of the rules and exit. It might seem at first that if the generated rule is partial, while there exists at least one tautological rule for the same triplet, exiting is undesirable, since the opportunity to render the triplet tautological might be missed. However, if a partial rule is applied, PROPAGATEBV returns unknown, thus PROPAGATEBV will be invoked again over the same triplet, hence one of the tautological rules will eventually be applied.

Let us consider the function GETC. It maintains a hash table for every triplet type with a non-empty set of rules. Each hash table entry contains a rule: its premise is mapped to its conclusion. GETC starts by looking for a conclusion for the given triplet in the corresponding hash table. If it is found, the algorithm acts based on the conclusion type. Namely, it returns active for an empty conclusion and throws a contradiction for a contradicting conclusion. For other conclusions, it merges the resulting bindings and returns tautological for a tautological conclusion and unknown for a partial conclusion.

If no rule is found in the hash table (line 28), GETC generates one. To generate a rule, GETC bit-blasts the skeleton to a new SAT solver instance Q. If Q is unsatisfiable, a contradictory rule is generated, otherwise the function generates a conclusion by invoking the auxiliary function CL to find maximal values for the conclusion. If the conclusion is interesting, the function checks whether the rule is tautological with another new SAT solver instance. The function then inserts the rule into the corresponding hash table and triggers the same behavior as if the newly generated rule was found in the hash table.

Algorithm 2 Rewriting for Bit-Vector 0-saturation

```
1: function PropagateBV(Triplet $t \equiv x = o(y,z)$)
2:     $r :=$ active
3:     if Both $y$ and $z$ are constants then Merge($x$, $o(y,z)$) & return tautological
4:     if $z \in \mathcal{RC}$ then $r :=$ GetC($t$, $i_x$, $i_y$, $z$) & if $r \neq$ active return $r$
5:     if $y \in \mathcal{RC}$ then $r :=$ GetC($t$, $i_x$, $y$, $i_z$) & if $r \neq$ active return $r$
6:     if $x \in \mathcal{RC}$ then $r :=$ GetC($t$, $x$, $i_y$, $i_z$) & if $r \neq$ active return $r$
7:     if $x \in \mathcal{RF}_y$ then $r :=$ GetC($t$, $x$, $i_y$, $i_z$) & if $r \neq$ active return $r$
8:     if $x \in \mathcal{RF}_z$ then $r :=$ GetC($t$, $x$, $i_y$, $i_z$) & if $r \neq$ active return $r$
9:     if $y \in \mathcal{RF}_z$ then $r :=$ GetC($t$, $i_x$, $y$, $i_z$) & if $r \neq$ active return $r$
10:    if $x, y \in \mathcal{RC}$ then $r :=$ GetC($t$, $x$, $y$, $i_z$) & if $r \neq$ active return $r$
11:    if $x \in \mathcal{RC}$ and $y \in \mathcal{RF}_z$ then $r :=$ GetC($t$, $x$, $y$, $i_z$) & if $r \neq$ active return $r$
12:    if $x \in \mathcal{RF}_y$ and $z \in \mathcal{RC}$ then $r :=$ GetC($t$, $x$, $i_y$, $z$) & if $r \neq$ active return $r$
13:    if $x \in \mathcal{RF}_z$ and $y \in \mathcal{RC}$ then $r :=$ GetC($t$, $x$, $y$, $i_z$) & if $r \neq$ active return $r$
14:    if $x \in \mathcal{RF}_y$ and $y \in \mathcal{RF}_z$ then $r :=$ GetC($t$, $x$, $y$, $i_z$)
15:    return $r$

16: function CL(SAT Instance $Q$; $\sigma \in \{\sigma_x, \sigma_y, \sigma_z\}$)
17:     return a maximal $\rho$, such that $Q \wedge (\sigma \neq \rho)$ is UNSAT, where $\rho$ must be
    x-value/y-value/z-value iff $\sigma$ is $\sigma_x/\sigma_y/\sigma_z$, respectively

18: function GetC(Triplet $t \equiv x = o(y,z)$ of type $= \{o, w\}$; Premise $p = \{\sigma_x, \sigma_y, \sigma_z\}$)
19:     if rules [type] [$p$] exists then
20:         $c :=$ rules [type] [$p$]
21:         if $c$ is empty then return active
22:         if $c$ is contradictory then throw contradiction
23:         $c$ must be of the form $\{\rho_x, \rho_y, \rho_z, \text{taut}\}$
24:         if $\rho_x \neq \sigma_x$ then Merge($x$, $\rho_x^{i_x \mapsto x}$)
25:         if $\rho_y \neq \sigma_y$ then Merge($y$, $\rho_y^{i_y \mapsto y}$)
26:         if $\rho_z \neq \sigma_z$ then Merge($z$, $\rho_z^{i_z \mapsto z}$)
27:         if taut = true then return tautological else return unknown
28:     else
29:         Bit-blast the skeleton $\sigma_x^{i_x \mapsto x} = o(\sigma_y^{i_y \mapsto y}, \sigma_z^{i_z \mapsto z})$ to a new SAT instance $Q$
30:         if $Q$ is unsatisfiable then
31:             rules [type] [$p$] $:= \bot$
32:         else
33:             $\rho_x :=$ CL($Q$, $\sigma_x$); $\rho_y :=$ CL($Q$, $\sigma_y$); $\rho_z :=$ CL($Q$, $\sigma_z$)
34:             if $\rho_x \neq \sigma_x$ or $\rho_y \neq \sigma_y$ or $\rho_z \neq \sigma_z$ then
35:                 Bit-blast $\sigma_x^{i_x \mapsto x} = \rho_x^{i_x \mapsto x} \wedge \sigma_y^{i_y \mapsto y} = \rho_y^{i_y \mapsto y} \wedge \sigma_z^{i_z \mapsto z} = \rho_z^{i_z \mapsto z} \wedge (\sigma_x^{i_x \mapsto x} \neq$
    $o(\sigma_y^{i_y \mapsto y}, \sigma_z^{i_z \mapsto z}))$ to a new SAT instance $R$
36:                 if $R$ is unsatisfiable then taut := true else taut := false
37:                 rules [type] [$p$] $:= \{\rho_x, \rho_y, \rho_z, \text{taut}\}$
38:             else
39:                 rules [type] [$p$] $:= \{\sigma_x, \sigma_y, \sigma_z, \text{false}\}$
40:         Go to line 20
```

4 Experimental Results

We implemented our new algorithm in Intel's eager bit-vector solver Hazel, which operates by invoking bit-vector preprocessing, followed by bit-blasting to CNF and SAT solving. Both base and new Hazel use the standard manual offline instance-generic DAG-based rewriting, where the novel automatic rewriting in new Hazel is applied after the manual rewriting (switching off manual rewriting in Hazel is impossible). We compared the performance of the new version of Hazel to base Hazel and the latest publicly available versions of the state-of-the-art SMT solvers Boolector [6] (version 1.6.0) and Mathsat [7] (version 5.2.10; SMT'11 competition configuration). We also gathered some statistics.

The benchmark families we used belong to the QF_BV category of SMT-LIB [5]. Since this category contains tens of thousands of benchmarks, we could not use all of them. We decided to pick all 23 families of the ASP sub-category, since, while these families are difficult and versatile, they contain a tractable number of benchmarks. In our analysis we skipped about 12% of the benchmarks that could not be solved by any of the four solvers within the time-out of 20 minutes. We used machines running Intel® Xeon® processors with 3Ghz CPU frequency and having 32Gb of memory. Detailed experimental results are available at [16].

Consider Table 2, which presents the results. Each row corresponds to one family. Column 1 contains the family name, abbreviated to the first three letters (except for GraphColouring and GraphPartitioning, which are represented as GC and GP, respectively). Column 2 contains the number of instances. Each pair of neighboring columns of the subsequent eight columns provides the overall run-time in seconds (where the time-out value was added to the run-time for unsolved instances) and the number of unsolved instances for the solver listed in the column heading. The best performance is highlighted.

Compare the performance of the new version of Hazel over the 23 families to that of the other solvers. New Hazel outperforms base Hazel on 20 families (that is, new Hazel either solves more instances or the same number of instances in less time on 20 families). New Hazel outperforms Mathsat on 21 families and Boolector on 14 families. Moreover, there are 10 families on which new Hazel significantly outperforms all the other solvers: it either solves more instances or is at least 2x faster. These results testify clearly that our approach can considerably boost the performance of modern SMT solvers.

The final seven columns of Table 2 contain statistics. Column 11 shows the average 0-saturation run-time, including automatic rule generation, as a percent of the overall run-time of new Hazel. One can see that the run-time of 0-saturation is negligible. Columns 12 and 13 demonstrate the impact of 0-saturation on the size of the CNF by showing the average percent of CNF clauses and variables, respectively, in new Hazel as compared to base Hazel (for example, f in Column 12 means that if base Hazel generated c clauses, then new Hazel generated $f/100 * c$ CNF clauses). Note that for the vast majority of families 0-saturation significantly reduces the number of CNF clauses and variables. One notable exception is the Wei family, where the number of CNF clauses is slightly greater when 0-saturation is applied. This can happen because 0-saturation may need to *create* variables, if required by the rule's conclusion (e.g., new neighbors may be created). Although 0-saturation did not reduce the number of clauses in the Wei case, it did simplify the CNF instances considerably, as the performance speed-up attests. Columns 14 and 15 provide the *average* number of interesting rules generated and applied, respectively, by new Hazel, while columns 16 and 17 provide the same data for the non-interesting rules. One can see that 0-saturation creates only few rules, but applies them very often, sometimes up to hundreds of thousands of times.

Finally, we report that our algorithm did not generate any contradictory or partial rules in our experiments.

Table 2. Performance Comparison and Statistics.

		Boolector		Mathsat		Base Hazel		New Hazel		0-sat	CNF Red		Intr Rules		Non-I Rules	
Fam	#	Time	Un	Time	Un	Time	Un	Time	Un	%Tm	%Cls	%Var	#	Applied	#	Applied
Dis	4	3582	2	2523	2	4800	4	**515**	**0**	0.7	49	25	7	1707424	10	915729
Sol	23	5825	1	24802	18	27600	23	**2200**	**1**	0.5	56	52	8	48037	10	15890
Lab	10	12000	10	12000	10	10342	8	**6996**	**0**	0.1	67	48	12	1070897	15	209523
Edg	29	30693	23	32391	24	25073	19	**5345**	**0**	0.2	61	56	3	322700	11	85153
Wei	29	5550	2	20228	10	25625	16	**2591**	**0**	0.1	101	100	6	6390	5	2940
Sud	8	7213	6	7213	6	5918	4	**3229**	**0**	0.4	63	67	6	156591	10	122519
GC	12	**2771**	**1**	5674	4	4952	4	3362	1	0.8	71	44	2	16077	9	8496
GP	7	1418	0	2939	2	2514	1	**205**	**0**	0.6	74	72	3	29798	13	6234
Fas	17	3869	0	6487	3	4272	2	**1152**	**0**	2.7	81	57	2	125189	5	4445
Ham	29	**271**	**0**	4036	3	3678	3	2694	1	0.7	81	71	5	10700	10	7134
Sok	29	872	0	15700	8	2662	0	**833**	**0**	0.7	23	20	11	45167	16	14044
Hie	12	1165	0	267	0	123	0	**56**	**0**	1.2	65	63	2	45234	9	26376
15P	15	477	0	2654	0	359	0	**168**	**0**	1.8	35	25	8	146586	11	18110
Han	15	1345	0	1584	0	200	0	**101**	**0**	1.9	63	48	10	136458	11	44241
Gen	29	1226	0	506	0	352	0	**220**	**0**	2.6	89	82	3	29794	12	11944
Cha	8	117	0	60	0	18	0	**12**	**0**	4.7	67	29	2	105654	9	35987
Kni	3	**67**	**0**	1347	1	1798	1	1280	1	0.3	70	65	9	17734	12	9014
Blo	29	**1023**	**0**	30521	22	16213	5	11737	5	0.0	67	61	2	13208	9	6037
Sch	29	**1053**	**0**	8061	4	6938	3	6643	3	2.7	65	23	8	61756	11	32450
Wir	19	**6330**	**0**	16511	12	8712	5	8567	5	0.6	85	63	9	172717	15	60664
Tra	29	**6449**	**0**	33339	26	34800	29	34800	29	0.0	73	80	6	20007	12	9811
Con	21	**294**	**0**	612	0	382	0	499	0	0.7	87	62	2	6538	4	651
Maz	29	**879**	**0**	2610	0	2518	0	3543	1	0.9	92	53	6.5	10009	3	1995

5 Conclusion

We have proposed a new preprocessing algorithm for bit-vector SMT solving: bit-vector 0-saturation with automatic rewriting rule generation. Applying our algorithm in Intel's SMT solver Hazel resulted in a substantial performance improvement over 23 ASP families from SMT-LIB. New Hazel outperforms the base version of Hazel on 20 families; it outperforms Mathsat on 21 families and Boolector on 14 families. Moreover, there are 10 families on which new Hazel outperforms base Hazel, Boolector, and Mathsat significantly: it either solves more instances or is at least 2x faster. Our approach can be improved further by extending it to generate more types of rewriting rules automatically.

References

1. Domagoj Babic. *Exploiting structure for scalable software verification.* Dissertation, The University Of British Columbia, 2008.
2. Sorav Bansal. *Peephole Superoptimization.* Dissertation, Stanford University, 2008.
3. Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In John Paul Shen and Margaret Martonosi, editors, *ASPLOS*, pages 394–403. ACM, 2006.
4. Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
5. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.
6. Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.
7. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.
8. Stephan Falke, Florian Merz, and Carsten Sinz. LLBMC: improved bounded model checking of C programs using LLVM. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 623–626. Springer, 2013.
9. Anders Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT.* Dissertation, University of Trento, 2010.
10. Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. Applying SMT in symbolic execution of microcode. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 121–128. IEEE, 2010.
11. Vijay Ganesh, Sergey Berezin, and David L. Dill. A Decision Procedure for Fixed-Width Bit-Vectors. Technical report, Computer Science Department, Stanford University, April 2005.
12. John Harrison. Stålmarck's algorithm as a HOL derived rule. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *TPHOLs*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234. Springer, 1996.
13. Michael Katelman and José Meseguer. vlogsl: A strategy language for simulation-based verification of hardware. In *Hardware and Software: Verification and Testing*, pages 129–145. Springer, 2011.
14. Filip Marić and Predrag Janičić. URBiVA: Uniform reduction to bit-vector arithmetic. In *Automated Reasoning*, pages 346–352. Springer, 2010.
15. Raphaël Michel, Arnaud Hubaux, Vijay Ganesh, and Patrick Heymans. An SMT-based approach to automated configuration. In *SMT Workshop 2012 10th International Workshop on Satisfiability Modulo Theories SMT-COMP 2012*, page 107, 2012.
16. Alexander Nadel. Detailed experimental results for bit-vector rewriting with automatic rule generation: https://drive.google.com/file/d/0B0zXW5t7in-fc0dmdXNVUWttVXc/edit?usp=sharing.
17. Jakob Nordström. Stålmarck's method versus resolution: A comparative theoretical study. Master's thesis, Stockholm University, Stockholm, Sweden, 2001.

18. Anthony Romano and Dawson Engler. Expression reduction from programs in a symbolic binary executor. In Ezio Bartocci and C. R. Ramakrishnan, editors, *SPIN*, volume 7976 of *Lecture Notes in Computer Science*, pages 301–319. Springer, 2013.
19. Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarcks's proof procedure for propositional logic. In Ganesh Gopalakrishnan and Phillip J. Windley, editors, *FMCAD*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99. Springer, 1998.
20. Alexander Hansen Trevor. *A constraint solver and its application to machine code test generation*. Dissertation, Dept. of Computing and Information Systems, The University of Melbourne, 2012.
21. Robert Wille, Daniel Große, Finn Haedicke, and Rolf Drechsler. SMT-based stimuli generation in the SystemC verification library. In *Advances in Design Methods from Modeling Languages for Embedded Systems and SoCs*, pages 227–244. Springer, 2010.