FMSD manuscript No.
(will be inserted by the editor)

Efficiently Solving Quantified Bit-Vector Formulas

Christoph M. Wintersteiger · Youssef Hamadi · Leonardo de Moura

Received: date / Accepted: date

Abstract In recent years, bit-precise reasoning has gained importance in hardware and software verification. Of renewed interest is the use of symbolic reasoning for synthesising loop invariants, ranking functions, or whole program fragments and hardware circuits. Solvers for the quantifier-free fragment of bit-vector logic exist and often rely on SAT solvers for efficiency. However, many techniques require quantifiers in bit-vector formulas to avoid an exponential blow-up during construction. Solvers for quantified formulas usually flatten the input to obtain a quantified Boolean formula, losing much of the word-level information in the formula. We present a new approach based on a set of effective word-level simplifications that are traditionally employed in automated theorem proving, heuristic quantifier instantiation methods used in SMT solvers, and model finding techniques based on skeletons/templates. Experimental results on two different types of benchmarks indicate that our method outperforms the traditional flattening approach by multiple orders of magnitude of runtime.

Keywords Theorem Proving · Satisfiability · SMT · Bit-Vectors · QBF

1 Introduction

The complexity of integrated circuits continues to grow at an exponential rate and so does the size of the verification and synthesis problems arising from the hardware design process. To tackle these problems, bit-precise decision procedures

Christoph M. Wintersteiger
Microsoft Research, Cambridge, UK
E-mail: cwinter@microsoft.com

Youssef Hamadi
Microsoft Research, Cambridge, UK
E-mail: youssefh@microsoft.com

Leonardo de Moura
Microsoft Research, Redmond, U.S.A.
E-mail: leonardo@microsoft.com

are a requirement and oftentimes the crucial ingredient that defines the efficiency of the verification process.

Recent years also saw an increase in the utility of bit-precise reasoning in the area of software verification where low-level languages like C or C++ are concerned. In both areas, hardware and software design, methods of automated synthesis (e.g., LTL synthesis [25]) become more and more tangible with the advent of powerful and efficient decision procedures for various logics, most notably SAT and SMT solvers. In practice, however, synthesis methods are often incomplete, bound to very specific application domains, or simply inefficient.

In the case of hardware, synthesis usually amounts to constructing a module that implements a specification [18,25], while for software this can take different shapes: inferring program invariants [14], finding ranking functions for termination analysis [8,26,31], program fragment synthesis [29], or constructing bug fixes following an error-description [30] are all instances of the general synthesis problem.

In this paper, we present a new approach to solving quantified bit-vector logic. This logic allows for a direct mapping of hardware and (finite-state) software verification problems and is thus ideally suited as an interface between the verification or synthesis tool and the decision procedure.

In many practically relevant applications, support for uninterpreted functions is not required and if this is the case, quantified bit-vector formulas can be reduced to quantified Boolean formulas (QBF). In practice however, QBF solvers face performance problems and they are usually not able to produce models for satisfiable formulas, which is crucial in synthesis applications. The same holds true for many automated theorem provers. SMT solvers on the other hand are efficient and produce models, but usually lack complete support for quantifiers.

The ideas in this paper combine techniques from automated theorem proving, SMT solving and synthesis algorithms. We propose a set of simplifications and rewriting techniques that transform the input into a set of equations that an SMT solver is able to solve efficiently. A model finding algorithm is then employed to refine a candidate model iteratively, while we use function or circuit templates to reduce the number of iterations required by the algorithm. Finally, we evaluate a prototype implementation of our algorithm on a set of hardware and software benchmarks, which indicate speedups of up to five orders of magnitude compared to flattening the input to QBF.

2 Background

We assume the usual notions and terminology of first order logic and model theory [15]. We are mainly interested in many-sorted languages and bit-vectors of different sizes correspond to different sorts. We assume that, for each bit-vector sort of size n, the equality $=_n$ is interpreted as the identity relation over bit-vectors of size n, i.e., $=_n(x, y) := x = y$, where x and y are bit-vectors of size n. The if-then-else (multiplexer) bit-vector term ite_n is interpreted as usual as $ite(true, t, e) = t$ and $ite(false, t, e) = e$. As a notational convention, we always omit the subscript specifying the size of the bit-vectors. We call 0-arity function symbols *constant* symbols, and 0-arity predicate symbols *propositions*. *Atoms*, *literals*, *clauses*, and *formulas* are defined in the usual way [15]. Terms, literals, clauses and formulas are called *ground* when no variable appears in them. A *sentence* is a formula in

which free variables do not occur. A *CNF formula* is a conjunction $C_1 \wedge \ldots \wedge C_n$ of clauses. We write CNF formulas as sets of clauses. We use a, b and c for constants, f and g for function symbols, p and q for predicate symbols, x, y and z for variables, C for clauses, φ for formulas, and t for terms. We use $x{:}\,n$ to denote that variable x is a bit-vector of size n. When the bit-vector size is not specified, it is implicitly assumed to be 32. We use $f{:}\,n_1, \ldots, n_k \rightarrow n_r$ to denote that function symbol f has arity k, argument bit-vectors have sizes $n_1, \ldots, n_k$, and the result bit-vector has size n_r. We use $\varphi[x_1, \ldots, x_n]$ to denote a formula that may contain variables $x_1, \ldots, x_n$, and similarly $t[x_1, \ldots, x_n]$ is defined for a term t. Where there is no confusion, we denote $\varphi[x_1, \ldots, x_n]$ by $\varphi[\overline{x}]$ and $t[x_1, \ldots, x_n]$ by $t[\overline{x}]$. In the rest of this paper, the difference between functions and predicates is trivial, and we will thus only discuss functions except at a few places.

We use the standard notion of a structure (interpretation). A structure that satisfies a formula F is said to be a model for F. A theory is a collection of first-order sentences. Interpreted symbols are those symbols whose interpretation is restricted to the models of a certain theory. We say a symbol is free or uninterpreted if its interpretation is not restricted by a theory. We use *BitVec* to denote the bit-vector theory. In this paper we assume the usual interpreted symbols for bit-vector theory: $+_n$, $*_n$, $concat_{m,n}$, $\leq_n$, 0_n, 1_n, ... Where there is no confusion, we omit the subscript specifying the actual size of the bit-vector.

A formula is *satisfiable* if and only if it has a model. A formula F is satisfiable modulo the theory *BitVec* if there is a model for $\{F\} \cup BitVec$.

3 Quantified Bit-Vector Formulas

A *Quantified Bit-Vector Formula* (QBVF) is a many sorted first-order logic formula where the sort of every variable is a bit-vector sort. The QBVF-satisfiability problem is the problem of deciding whether a QBVF is satisfiable modulo the theory of bit-vectors. This problem is decidable because every universal (existential) quantifier can be expanded into a conjunction (disjunction) of potentially exponential, but finite size. A distinguishing feature in QBVF is the support for uninterpreted function and predicate symbols, which, for example, allows for an encoding of the theory of arrays over bit-vectors.

Example 1 The formula $\forall x : 32\ \exists y : 32\ .\ y = x +_{32} 1_{32}$ is satisfiable, because the $+_{32}$ operation overflows, i.e., at $x = (2^{32} - 1)$, we have $x +_{32} 1_{32} = 0_{32}$. On the other hand, $\exists x : 16\ \forall y : 16\ .\ x <_{16} x *_{16} y$ is not satisfiable because y may be 0_{16} or 1_{16}. Note that this second formula becomes satisfiable when we change the $<_{16}$ operator to the weaker $\leq_{16}$ which allows both sides to be equal.

Quantified *Boolean* formulas (QBF) are a generalisation of Boolean formulas, where quantifiers can be applied to each variable. Deciding a QBF is a PSPACE-complete problem. Note that any QBF problem can be easily encoded in QBVF by using bit-vectors of size 1. The converse is not true, QBVF is more expressive than QBF. For instance, uninterpreted function symbols can be used to simulate non-linear quantifier prefixes.[1] The *effectively propositional (EPR)* fragment of

[1] We mean quantifier prefixes which cannot be written in a linear fashion, e.g., $\forall x \exists y \ldots$, where y may *not* depend on x. In the general case, a 'graph' of quantifiers is required for such constraints.

first-order logic comprises formulas of the form $\exists^*\forall^*\varphi$, where φ is a quantifier-free formula with predicates but without function symbols. EPR is a decidable fragment because the Herbrand universe of a EPR formula is always finite. The satisfiability problem for EPR is known to be NEXPTIME-complete [20]; a result which allows us to show that the QBVF-satisfiability problem is of the same complexity:

Theorem 1 *The satisfiability problem for QBVF is NEXPTIME-complete.*

Proof The proof consists in showing that there is a polynomial reduction from QBVF to EPR and vice-versa.

QBVF $\Rightarrow$ EPR. Given a QBV formula φ, w.l.o.g. we assume φ is in CNF. The first step is to flatten every clause in φ. The idea is to avoid nested terms by introducing auxiliary variables. Given a clause $\forall\overline{x}.\ C[t]$, where t is a nested term, we convert it to $\forall\overline{x}, y.\ y \neq t \vee C[y]$. Flattening is applied until all literals in a clause are *shallow*. For example, the clause $\forall x_1, x_2.\ f(x_1, g(x_2)) \leq g(x_1)$ is reduced to

$$\begin{aligned}\forall x_1, x_2, y_1, y_2, y_3.\ & y_1 \neq g(x_2) \vee y_2 \neq f(x_1, y_1) \vee \\ & y_3 \neq g(x_1) \vee y_2 \leq y_3\ .\end{aligned}$$

Next, for each uninterpreted function f where the codomain is a bit-vector of size n, we create n predicates $p_{f_1}, \ldots, p_{f_n}$. Each bit-vector variable and constant is broken into bits. A disequality of the form $x \neq f(y_1, \ldots, y_m)$ is encoded as

$$\begin{aligned}&((x_1 = \top)\ \mathit{xor}\ p_{f_1}(y_1, \ldots, y_m)) \vee \\ &\ldots \\ &((x_n = \top)\ \mathit{xor}\ p_{f_n}(y_1, \ldots, y_m))\ .\end{aligned}$$

To achieve an encoding of polynomial size, we assume that the number n is provided in a unary encoding in the QBVF. This is not a problem in practical applications since n is usually very small (e.g., $n \leq 64$). Alternatively, assuming the occurrence of at least one bit-vector constant of size equal to the largest n in the QBVF ensures that our encoding will be of polynomial size.

Other atoms are encoded in a similar fashion. We add two special constants $\bot$ and $\top$, add the axiom $\bot \neq \top$, and for each new bit constant c, we add the clause $c = \bot \vee c = \top$. For example, assume that in

$$(\forall x.\ f(f(x)) = 0) \wedge f(a) = 2$$

all sorts are bit-vectors of size 2. After flattening, we obtain

$$(\forall x, y.\ y \neq f(x) \vee f(y) = 0) \wedge f(a) = 2\ .$$

Then, after bit-blasting, we have

$$\begin{aligned}&(\forall x_1, x_2, y_1, y_2.\ ((y_1 = \top)\ \mathit{xor}\ p_{f_1}(x_1, x_2)) \vee \\ &\qquad\qquad ((y_2 = \top)\ \mathit{xor}\ p_{f_2}(x_1, x_2)) \vee \\ &\qquad\qquad (\neg p_{f_1}(y_1, y_2) \wedge \neg p_{f_2}(y_1, y_2))) \wedge \\ &\neg p_{f_1}(a_1, a_2)\ \wedge\ p_{f_2}(a_1, a_2) \wedge \\ &(a_1 = \top \vee a_1 = \bot) \wedge \\ &(a_2 = \top \vee a_2 = \bot) \wedge \\ &\top \neq \bot\ .\end{aligned}$$

EPR $\Rightarrow$ *QBVF.* Any satisfiable EPR formula has a finite Herbrand model. Moreover, a formula containing n constants has a model with a universe of size at most n. In principle, we just need to use a bit-vector sort of size $\lceil log_2 n \rceil$. The main problem in this approach is that the EPR formula may contain cardinality constraints such as $\forall x.\ x = a_1 \vee \ldots \vee x = a_m$. For example, this clause is only satisfiable in a model with a universe with size at most m. Now, suppose we have a formula φ with n constants and containing a cardinality constraint limiting the universe size to m. If $m < \lceil log_2 n \rceil$, then the QBV formula

$$\forall x : \lceil log_2 n \rceil.\ x = a_1 \vee \ldots \vee x = a_m$$

is equivalent to *false*. This problem can be avoided by using an approach found in several EPR solvers that do not have support for $=$. These solvers use the fact that any EPR formula φ containing $=$ is equisatisfiable to another EPR formula φ' that does not contain $=$. The basic idea is to replace $=$ with a new binary predicate *isEq*, and include the axioms of equality for it:

$$\begin{array}{ll}
\forall x. & isEq(x,x) \\
\forall x,y. & \neg isEq(x,y) \vee isEq(y,x) \\
\forall x,y,z. & \neg isEq(x,y) \vee \neg isEq(y,z) \vee isEq(x,z) \\
\forall \overline{x},\overline{y}. & \neg isEq(x_1,y_1) \vee \ldots \vee \neg isEq(x_n,y_n) \vee \neg p(\overline{x}) \vee p(\overline{y})\ .
\end{array}$$

In fact the last axiom is an axiom scheme; we need one of them for each predicate p in the formula φ. □

QBVF can be used to compactly encode many practically relevant verification and synthesis problems. In hardware verification, a fixpoint check consists in deciding whether k unwindings of a circuit are enough to reach all states of the system. To check this, two copies of the k unwindings are used: Let $T[x,x']$ be a formula encoding the transition relation and $I[x]$ a formula encoding the initial states of a circuit. Furthermore, we define

$$T^k[x,x'] \equiv T[x,x_0] \wedge \bigwedge_{i=1}^{k-1} T[x_{i-1},x_i] \wedge T[x_{k-1},x']\ .$$

Then a fixpoint check for k unwindings corresponds to the QBV formula

$$\forall x,x'\ .\ I[x] \wedge T^k[x,x'] \rightarrow \exists y,y'\ .I[y] \wedge T^{k-1}[y,y']\ ,$$

where x, x', y, and y' are (usually large) bit-vectors.

Of renewed interest is the use of symbolic reasoning for synthesising code [29], loop invariants [7,14] and ranking functions for finite-state programs [8]. All these applications can be easily encoded in QBVF. To illustrate these ideas, consider the following abstract program:

pre
while (c) { T }
post

In the loop invariant synthesis problem, we want to synthesise a predicate I that can be used to show that *post* holds after execution of the *while-loop*. Let, $pre[x]$ be a formula encoding the set of states reachable before the beginning of

the loop, $c[x]$ be the encoding of the entry condition, $T[x, x']$ be the transition relation, and $post[x]$ be the encoding of the property we want to prove. Then, a suitable loop invariant exists if the following QBV formula is satisfiable:

$$\begin{array}{l} \forall x.\ pre[x] \rightarrow I(x) \ \wedge \\ \forall x, x'.\ I(x) \wedge c[x] \wedge T[x, x'] \rightarrow I(x') \ \wedge \\ \forall x.\ I(x) \wedge \neg c[x] \rightarrow post[x]\ . \end{array}$$

An actual invariant can be extracted from any model that satisfies this formula.

Similarly, in the ranking function synthesis problem, we want to synthesise a function *rank* that decreases after each loop iteration and that is bounded from below. The idea is to use this function to show that a particular loop in the program always terminates. This problem can be encoded as the following QBVF satisfiability problem:

$$\begin{array}{l} \forall x.\ rank(x) \geq 0 \ \wedge \\ \forall x, x'.\ c[x] \wedge T[x, x'] \rightarrow rank(x') < rank(x)\ . \end{array}$$

Note that the general case of this encoding requires uninterpreted functions. The call to *rank* can not be replaced with an existentially quantified variable, as it is impossible to express the correct dependencies in a linear quantifier prefix.

3.1 Encoding bit-vector arrays

The extensional theory of arrays is characterised by the following axioms:

$$\begin{array}{l} \forall x, y, z.\ select(store(x, y, z), y) = z, \\ \forall x, y_1, y_2, z.\ y_1 = y_2 \vee select(store(x, y_1, z), y_2) = select(x, y_2), \\ \forall x_1, x_2.\ (\forall y.\ select(x_1, y) = select(x_2, y)) \rightarrow x_1 = x_2\ . \end{array}$$

Arrays of bit-vectors can be easily encoded in QBVF using quantifiers and uninterpreted function symbols. For every array term t we create a fresh uninterpreted function f_t. Then, we replace terms of the form $select(t, i)$ with $f_t(i)$. For every term t of the form $store(s, i, v)$, we add the universal formula $\forall x.\ x = i \vee f_t(x) = f_s(x)$, and the ground atom $f_t(i) = v$. Finally, we replace equations of the form $t = s$, where t and s are arrays, with $\forall x. f_t(x) = f_s(x)$.

Example 2 Let F be the formula $select(b, j) \neq 0 \wedge a = s$, where s is a term of the form $store(b, i + 1, 0)$. Then, F is encoded in QBVF as

$$f_b(j) \neq 0 \wedge (\forall x.\ x = i + 1 \vee f_s(x) = f_b(x)) \wedge f_s(i + 1) = 0 \wedge (\forall x.\ f_a(x) = f_s(x))\ .$$

4 Solving QBVF

In this section, we describe a QBVF solver based on ideas from first-order theorem proving, SMT solving and synthesis tools. First, we present a set of simplifications and rewriting rules that help to greatly reduce the size and complexity of typical QBV formulas. Then, we describe how to check whether a given model satisfies a QBVF and how to use this to construct new models, using templates to speed up the process (sometimes exponentially).

4.1 Simplifications & Rewriting

Modern first-order theorem provers spend a great part of their time in simplifying or contracting operations. These operations are inferences that remove or modify existing formulas. Our QBVF solver implements several simplification or contraction rules found in first-order provers. We also propose new rules that are particularly useful in our application domain.

4.1.1 Miniscoping

Miniscoping is a well-known technique for minimising the scope of quantifiers [15]. We apply this transformation after converting the formula to negation normal form (NNF, [15]). The basic idea is to distribute universal (existential) quantifiers over conjunctions (disjunctions). For the universal case, we have:

$$(\forall \overline{x}.F[\overline{x}] \wedge G[\overline{x}]) \Longrightarrow (\forall \overline{x}.F[\overline{x}]) \wedge (\forall \overline{x}.G[\overline{x}])$$

This transformation is particularly important in our context because it increases the applicability of rules based on rewriting and macros. We may limit the scope of a quantifier if a sub-formula does not contain the quantified variable, using

$$(\forall \overline{x}.F[\overline{x}] \vee G) \Longrightarrow (\forall \overline{x}.F[\overline{x}]) \vee G \ ,$$

when G does not contain x, and the corresponding rule for existential quantifiers.

4.1.2 Skolemization

Similarly to first-order theorem provers, in our solver, existentially quantified variables are eliminated using *Skolemization* [15]. A formula $\forall x.\ \exists y.\ \neg p(x) \vee q(x,y)$ is converted into the equisatisfiable formula $\forall x.\ \neg p(x) \vee q(x, f_y(x))$, where f_y is a fresh function symbol.

4.1.3 A conjunction of universally quantified formulas

After conversion to NNF, miniscoping and skolemization, the QBV formula is written as a conjunction of universally quantified formulas: $(\forall \overline{x}.\ \varphi_1[\overline{x}]) \wedge \ldots \wedge (\forall \overline{x}.\ \varphi_n[\overline{x}])$. This form is very similar to that used in first-order theorem provers. However, we do not require each $\varphi_i[\overline{x}]$ to be a clause. Note that some of the conjuncts may be ground, i.e., $\overline{x}$ may be empty.

4.1.4 Destructive Equality Resolution (DER)

DER allows us to solve a negative equality literal by applying the transformation

$$(\forall x, \overline{y}.\ x \neq t \vee \varphi[x, \overline{y}]) \Longrightarrow (\forall \overline{y}.\ \varphi[t, \overline{y}]) \ ,$$

where t does not contain x. For example, using DER, the formula $\forall x, y.\ x \neq f(y) \vee g(x,y) \leq 0$ is simplified to $\forall y.\ g(f(y), y) \leq 0$. DER is essentially an equality substitution rule. This becomes clear when we write the clause on the left-hand-side using an implication: $\forall x, \overline{y}.\ x = t \rightarrow \varphi[x, \overline{y}]$. It is straightforward to implement

DER; a naive implementation eliminates a single variable at a time. In our experiments, we observed that this naive implementation was a bottleneck in benchmarks where hundreds of variables could be eliminated. The natural solution is to eliminate as many variables simultaneously as possible. The only complication in this approach is that some of the variables being eliminated may *depend* on each other. We say a variable x *directly depends* on y in DER, when there is a literal $x \neq t[y]$. In general we are presented with a formula of the form

$$\forall x_1, \ldots, x_n, \overline{y}.\ x_1 \neq t_1 \vee \ldots \vee x_n \neq t_n \vee \varphi[x_1, \ldots, x_n, \overline{y}]\ ,$$

where each x_i may depend on variables x_j, $j \neq i$. First, we build a dependency graph G where the nodes are the variables x_i, and G contains an edge from x_i to x_j whenever x_j depends on x_i. Next, we perform a topological sort on G, and whenever a cycle is detected while visiting a node x_i, we remove x_i from G and move the corresponding $x_i \neq t_i$ to $\varphi[x_1, \ldots, x_n, \overline{y}]$. Finally, we use the variable order $x_{k_1}, \ldots, x_{k_m}$ $(m \leq n)$ produced by the topological sort to apply DER simultaneously. Let θ be a *substitution*, i.e., a mapping from variables to terms. Initially, θ is empty. For each variable x_{k_i} we first apply θ to t_{k_i} producing t'_{k_i}, and then update $\theta := \theta \cup \{x_{k_i} \mapsto t'_{k_i}\}$. After all variables x_{k_i} were processed, we apply the resulting substitution θ to $\varphi[x_1, \ldots, x_n, \overline{y}]$.

As a final remark, the applicability of DER can be increased using theory solvers. The idea is to rewrite inequalities of the form $t_1[x, \overline{y}] \neq t_2[x, \overline{y}]$, containing a universal variable x, into $x \neq t'[\overline{y}]$. This rewriting step is essentially equivalent to a theory solving step, where $t_1[x, \overline{y}] = t_2[x, \overline{y}]$ is solved for x. In the case of linear bit-vector equations, this can be achieved when the coefficient of x is odd [10].

4.1.5 Rewriting

The idea of using rewriting for equational reasoning is not new. It traces back to the work developed in the context of Knuth-Bendix completion [19]. The basic idea is to use unit clauses of the form $\forall \overline{x}.\ t[\overline{x}] = r[\overline{x}]$ as rewrite rules $t[\overline{x}] \rightsquigarrow r[\overline{x}]$, when $t[\overline{x}]$ is "bigger than" $r[\overline{x}]$. Any instance $t[\overline{s}]$ of $t[\overline{x}]$ is then replaced by $r[\overline{s}]$. For example, in the formula

$$(\forall x.\ f(x, a) = x)\ \wedge\ f(h(b), a) \geq 0\ ,$$

the quantifier can be used as the rewrite rule $f(x, a) \rightsquigarrow x$. Therefore, the term $f(h(b), a) \geq 0$ can be simplified to $h(b) \geq 0$, producing the new formula

$$(\forall x.\ f(x, a) = x)\ \wedge\ h(b) \geq 0\ .$$

We observed that rewriting is quite effective in many QBVF benchmarks, in particular, in hardware fixpoint check problems. Our goal is to use rewriting as an incomplete simplification technique. Thus, we are not interested in computing critical pairs or generating a confluent rewrite system. First-order theorem provers use sophisticated *term orderings* to orient the equations $t[\overline{x}] = r[\overline{x}]$ (see, e.g., [15]). We found that any term ordering, where interpreted symbols (e.g., +, *) are considered "small", suffices for our purposes. This can be realised, for instance, using a Knuth-Bendix Ordering where the weight of interpreted symbols is set to zero. The basic idea of this heuristic is to replace uninterpreted symbols with interpreted

ones.[2] For example, using $f(x) \rightsquigarrow 2x+1$, we can simplify $f(a)-a$ to $2a+1-a$, and then apply a bit-vector rewriting rule to further reduce it to $a+1$. We use a very simple approach where $t[\overline{x}]$ is considered bigger than $r[\overline{x}]$ if all universal variables in $r[\overline{x}]$ occur in $t[\overline{x}]$ and the number of uninterpreted symbols in $t[\overline{x}]$ is bigger than $r[\overline{x}]$. The basic idea is to use rewriting to replace uninterpreted symbols with interpreted ones. For example, we orient the equation $f(x) = 2x+1$ as $f(x) \rightarrow 2x+1$ instead of $2x+1 \rightarrow f(x)$ although the term $2x+1$ is syntactically bigger than $f(x)$. The idea behind this heuristic is to increase the applicability of theory specific rewriting rules. For example, using $f(x) \rightarrow 2x+1$, we can simplify $f(a)-a$ to $2a+1-a$, and then apply a bit-vector rewriting rule and reduce it to $a+1$.

4.1.6 Macros & Quasi-Macros

A *macro* is a unit clause of the form $\forall \overline{x}.\ f(\overline{x}) = t[\overline{x}]$, where f does not occur in t. Macros can be eliminated from QBV formulas by simply replacing any term of the form $f(\overline{r})$ with $t[\overline{r}]$. Any model for the resultant formula can be extended to a model that also satisfies $\forall \overline{x}.\ f(\overline{x}) = t[\overline{x}]$. For example, consider the formula

$$(\forall x.\ f(x) = x + a) \wedge f(b) > b\ .$$

After macro expansion, this formula is reduced to the equisatisfiable formula $b+a > b$. The interpretation $a \mapsto 1,\ b \mapsto 0$ is a model for this formula. This interpretation can be extended to

$$f(x) \mapsto x+1,\ \ a \mapsto 1,\ \ b \mapsto 0\ ,$$

which is a model for the original formula. This particular way to represent models is described in more detail in section 4.2.

A *quasi-macro* is a unit clause of the form

$$\forall \overline{x}.f(t_1[\overline{x}], \ldots, t_m[\overline{x}]) = r[\overline{x}]\ ,$$

where f does not occur in $r[\overline{x}]$, $f(t_1[\overline{x}], \ldots, t_m[\overline{x}])$ contains all $\overline{x}$ variables, and the following system of equations can be solved for $x_1, \ldots, x_n$

$$y_1 = t_1[\overline{x}],\ \ \ldots,\ \ y_m = t_m[\overline{x}]\ ,$$

where $y_1, \ldots, y_m$ are new variables. A solution of this system is a substitution

$$\theta : x_1 \mapsto s_1[\overline{y}],\ \ \ldots,\ \ x_n \mapsto s_n[\overline{y}]\ .$$

We use the notation $\varphi \downarrow \theta$ to represent the application of the substitution θ to the formula φ. Then, the *quasi-macro* can be replaced with the *macro*

$$\forall \overline{y}.f(\overline{y}) = ite(\bigwedge_i y_i = t_i[\overline{x}], r[\overline{x}], f'(\overline{y})) \downarrow \theta$$

where f' is a fresh function symbol. Intuitively, the new formula is saying that when the arguments of f are of the form $t_i[\overline{x}]$, then the result should be $r[\overline{x}]$, otherwise the value is not specified. After this transformation, the quasi-macro is in fact a macro and the quantifier can be eliminated using macro expansion.

[2] This reduces the search-space the solver has to traverse in the worst-case and therefore improves solver performance.

Example 3 The unit-clause

$$\forall x. f(x+1, x-1) = x$$

is a quasi-macro, because the system $y_1 = x+1,\ y_2 = x-1$ can be solved for x. A possible solution is the substitution $\theta = \{x \mapsto y_1 - 1\}$. Thus, we can transform this quasi-macro into the macro:

$$\forall y_1, y_2.\ f(y_1, y_2) = ite(y_1 = x+1 \wedge y_2 = x-1,\\ x,\ f'(y_1, y_2)) \downarrow \theta\ .$$

After applying the substitution θ and simplifying the formula, we obtain

$$\forall y_1, y_2.\ f(y_1, y_2) = ite(y_2 = y_1 - 2,\ y_1 - 1,\ f'(y_1, y_2))\ .$$

In our experiments, we observed that the solvability condition is trivially satisfied in many instances, because all variables $\overline{x}$ are actual arguments of f. Assume that variable x_i is the k_i-th argument of f. Then, the substitution θ is of the form $\{x_1 \mapsto y_{k_1}, \ldots, x_n \mapsto y_{k_n}\}$. For example, in many benchmarks we found quasi-macros that are larger variations of

$$\forall x_1, x_2.\ f(x_1,\ x_1 + x_2,\ x_2) = r[x_1, x_2]\ .$$

4.1.7 Function Argument Discrimination (FAD)

We have observed that after applying DER the i-th argument of many function applications is often a concrete bit-vector value such as: 0, 1, 2, etc. For any function symbol f and QBV formula φ, the following macro can be conjoined with φ while preserving satisfiability:

$$\forall x, \overline{y}.\ f(x, \overline{y}) = ite(x = v, f_v(\overline{y}), f'(x, \overline{y}))\ ,$$

where f_v and f' are fresh function symbols, and v is a bit-vector value. Now, suppose that the first argument of all f-applications are bit-vector values. The macro above will reduce $f(v', \bar{t})$ to $f_v(\bar{t})$ when $v = v'$, and $f'(v', \bar{t})$ otherwise. The transformation can be applied again to the f' applications if their first argument is again a bit-vector value.

Example 4 Let φ be the formula

$$(\forall x.\ f(1, x, 0) \geq x) \wedge\\ f(0, a, 1) < f(1, b, 0)\ \wedge\ f(0, c, 1) = 0\ \wedge\ c = a\ .$$

Applying FAD twice (for the values 0 and 1) on the first argument of f, we obtain

$$(\forall x.\ f_1(x, 0) \geq x) \wedge\\ f_0(a, 1) < f_1(b, 0)\ \wedge\ f_0(c, 1) = 0\ \wedge\ c = a\ .$$

Applying FAD for the third argument of f_1 and f_0 results in

$$(\forall x.\ f_{1,0}(x) \geq x) \wedge\\ f_{0,1}(a) < f_{1,0}(b)\ \wedge\ f_{0,1}(c) = 0\ \wedge\ c = a\ .$$

Since FAD is based on macro definitions, the infrastructure used for constructing interpretations for macros may be used to build an interpretation for f based on the interpretations of $f_{1,0}$ and $f_{0,1}$.

4.1.8 Other simplifications

As many other SMT solvers for bit-vector theory [2,5,6], our QBVF solver implements several bit-vector specific rewriting and simplification rules such as $a - a \Longrightarrow 0$. These rules have been proved to be very effective in solving quantifier-free bit-vector benchmarks, and this is also the case for the quantified case.

From now on, we assume there is a procedure `Simplify` that, given a QBV formula φ, converts it into negation normal form, then applies miniscoping, skolemization, and then applies the other simplification described in this section up to saturation.

4.2 Model Checking Quantifiers

Given a structure M, it is useful to have a procedure `MC` that checks whether M satisfies a universally quantified formula φ or not. We say `MC` is a *model checking procedure*. Before we describe how `MC` can be constructed, let us take a look at how structures are encoded in our approach. We use BV to denote the structure that assigns the usual interpretation to the (interpreted) symbols of the bit-vector theory (e.g., $+$, $*$, *concat*, etc). In our approach, the structures M are based on BV. We use $|BV|_n$ to denote the interpretation of the sort of bit-vectors of size n. By slight abuse of notation, the elements of $|BV|_n$ are $\{0_n, 1_n, \ldots, 2_n^{n-1}\}$. Again, where there is no confusion, we omit the subscript. The interpretation of an arbitrary term t in a structure M is denoted by $M[\![t]\!]$, and is defined in the standard way. We use $M\{x \mapsto v\}$ to denote a structure where the variable x is interpreted as the value v, and all other variables, function and predicate symbols have the same interpretation as in M. That is, $M\{x \mapsto v\}(x) = v$. For example, $BV\{x \mapsto 1\}[\![2 * x + 1]\!] = 3$. As usual, $M\{\overline{x} \mapsto \overline{v}\}$ denotes $M\{x_1 \mapsto v_1\}\{x_2 \mapsto v_2\} \ldots \{x_n \mapsto v_n\}$.

For each uninterpreted constant c that is a bit-vector of size n, the interpretation $M(c)$ is an element of $|BV|_n$. For each uninterpreted function (predicate) $f{:}\, n_1, \ldots, n_k \rightarrow n_r$ of arity k, the interpretation $M(f)$ is a term $t_f[x_1, \ldots, x_k]$, which contains only interpreted symbols and the free variables $x_1 : n_1, \ldots, x_k : n_k$. The interpretation $M(f)$ can be viewed as a *function definition*, where for all $\overline{v}$ in $|BV|_{n_1} \times \ldots \times |BV|_{n_k}$, $M(f)(\overline{v}) = BV\{\overline{x} \mapsto \overline{v}\}[\![t_f[\overline{x}]]\!]$.

Example 5 (Model representation) Let φ_a be the following formula:

$$
\begin{array}{l}
(\forall x.\ \neg(x \geq 0) \vee f(x) < x) \wedge \\
(\forall x.\ \neg(x < 0) \vee f(x) > x + 1) \wedge \\
f(a) > b \ \wedge \ b > a + 1\ .
\end{array}
$$

Then the interpretation

$$
\begin{array}{rl}
M_a := \{ & f(x) \mapsto \mathit{ite}(x \geq 0, x - 1, x + 3), \\
& a \mapsto -1, \\
& b \mapsto 1\}
\end{array}
$$

is a model for φ_a. For instance, we have $M[\![f(a)]\!] = 2$.

Usually, SMT solvers represent the interpretation of uninterpreted function symbols as finite *function graphs* (i.e., lookup tables). A function graph is an explicit representation that shows the value of the function for a finite (and relatively small) number of points. For example, let the function graph $\{0 \mapsto 1,\ 2 \mapsto 3,\ \mathit{else} \mapsto 4\}$ be the interpretation of the function symbol g. It states that the value of the function g at 0 is 1, at 2 it is 3, and for all other values it is 4. Any function graph can be encoded using *ite* terms. For example, the function graph above can be encoded as $g(x) \mapsto \mathit{ite}(x = 0, 1, \mathit{ite}(x = 2, 3, 4))$. Our approach for encoding interpretations is *symbolic* and potentially allows for an exponentially more succinct representation. For example, assuming f is a function from bit-vectors of size 32, the interpretation $f(x) \mapsto \mathit{ite}(x \geq 0, x - 1, x + 3)$ would correspond to a very large function graph.

When models are encoded in this fashion, it is straightforward to check whether a universally quantified formula $\forall \overline{x}.\ \varphi[\overline{x}]$ is satisfied by a structure M [11]. Let $\varphi^M[\overline{x}]$ be the formula obtained from $\varphi[\overline{x}]$ by replacing any term $f(\overline{r})$ with $M[\![f(\overline{r})]\!]$, for every uninterpreted function symbol f. A structure M satisfies $\forall \overline{x}.\ \varphi[\overline{x}]$ if and only if $\neg\varphi^M[\overline{s}]$ is unsatisfiable, where $\overline{s}$ is a tuple of fresh constant symbols.

Example 6 For instance, in Example 5, the structure M_a satisfies $\forall x.\ \neg(x \geq 0) \vee f(x) < x$ because

$$s \geq 0 \wedge \neg(\mathit{ite}(s \geq 0, s - 1, s + 3) < s)$$

is unsatisfiable. Let M_b be a structure identical to M_a in Example 5, but where the interpretation $M_b(f)$ of f is $x + 2$. M_b does not satisfy $\forall x.\ \neg(x \geq 0) \vee f(x) < x$ in φ_a because the formula $s \geq 0 \wedge \neg(s + 2 < s)$ is satisfiable, e.g., by $s \mapsto 0$. The assignment $s \mapsto 0$ is a *counter-example* for M_b being a model for φ_a.

The model-checking procedure `MC` expects two arguments, a universally quantified formula $\forall \overline{x}.\ \varphi[\overline{x}]$ and a structure M. It returns $\top$ if the structure satisfies $\forall \overline{x}.\ \varphi[\overline{x}]$ and a non-empty, finite set V of counter-examples otherwise. Each counter-example is a tuple of bit-vector values $\overline{v}$ s.t. $M\{\overline{x} \mapsto \overline{v}\}[\![\varphi[\overline{x}]]\!]$ evaluates to *false*.

4.3 Template Based Model Finding

In principle, the verification and synthesis problems described in section 3 can be attacked by any SMT solver that supports universally quantified formulas, and that is capable of producing models. Unfortunately, to the best of our knowledge, no SMT solver supports complete treatment of universally quantified formulas, even if the variables range over finite domains such as bit-vectors. On satisfiable instances, they will often not terminate or give up. On some unsatisfiable instances, SMT solvers may terminate using *heuristic-quantifier instantiation* [23].

It is not surprising that standard SMT solvers cannot handle these problems; the search space is simply too large. Synthesis tools based on automated reasoning try to constrain the search space using *templates*. For example, in the ranking function synthesis problem, the synthesis tool may limit the search to functions that are linear combinations of the inputs. This simple idea immediately transfers to QBVF solvers. In the context of a QBVF solver, a template is just an expression $t[\overline{x}, \overline{c}]$ containing free variables $\overline{x}$, interpreted symbols, and fresh constants $\overline{c}$. Given a tuple of bit-vector values $\overline{v}$, we say $t[\overline{x}, \overline{v}]$ is an *instance* of the template $t[\overline{x}, \overline{c}]$. A template can also viewed as a *parametric* function definition. For example, the

template $ax + b$, where a and b are fresh constants, may be used to guide the search for an interpretation for unary function symbols. The expressions $x + 1$ $(a \mapsto 1, b \mapsto 1)$ and $2x$ $(a \mapsto 2, b \mapsto 0)$ are instances of this template.

We say a *template binding* for a formula φ is a mapping from uninterpreted function (predicate) symbols f_i, occurring in φ, to templates $t_i[\overline{x}, \overline{c}]$. Conceptually, one template per uninterpreted symbol is enough. If we want to consider two different templates $t_1[\overline{x}, \overline{c_1}]$ and $t_2[\overline{x}, \overline{c_2}]$ for an uninterpreted symbol f, we can simply combine them in a single template $t'[\overline{x}, (\overline{c_1}, \overline{c_2}, c)] \equiv ite(c = 1, t_1[\overline{x}, \overline{c_1}], t_2[\overline{x}, \overline{c_2}])$, where c is a new fresh constant. This approach can be extended to construct templates that are combinations of smaller "instructions" that can be combined to construct a template for the desired class of functions.

Without loss of generality, let us assume that φ contains only one uninterpreted function symbol f. So, a template based model finder is a procedure `TMF` that given a ground formula φ and a template binding $\texttt{TB} = \{f \mapsto t[\overline{x}, \overline{c}]\}$, returns a structure M for φ such that the interpretation of f is $t[\overline{x}, \overline{v}]$ for some bit-vector tuple $\overline{v}$ if such a structure exists. `TMF` returns $\perp$ otherwise. Since we assume φ is a ground formula, a standard SMT solver can be used to implement `TMF`. We just need to check whether

$$\varphi \wedge \bigwedge_{f(\overline{r}) \in \varphi} f(\overline{r}) = t[\overline{r}, \overline{c}]$$

is satisfiable. If this is the case, the model produced by the SMT solver will assign values to the fresh constants $\overline{c}$ in the template $t[\overline{x}, \overline{c}]$. When $\texttt{TMF}(\varphi, \texttt{TB})$ succeeds we say φ is satisfiable *modulo* `TB`.

Example 7 (Template Based Model Finding) Let φ be the formula

$$f(a_1) \geq 10 \wedge f(a_2) \geq 100 \wedge f(a_3) \geq 1000 \wedge$$
$$a_1 = 0 \wedge a_2 = 1 \wedge a_3 = 2$$

and the template binding `TB` be $\{f \mapsto c_1 x + c_2\}$. Then, the corresponding satisfiability query is

$$f(a_1) \geq 10 \wedge f(a_2) \geq 100 \wedge f(a_3) \geq 1000 \wedge$$
$$a_1 = 0 \wedge a_2 = 1 \wedge a_3 = 2 \wedge$$
$$f(a_1) = c_1 a_1 + c_2 \wedge f(a_2) = c_1 a_2 + c_2 \wedge$$
$$f(a_3) = c_1 a_3 + c_2 \ .$$

The formula above is satisfiable, e.g., by the assignment $c_1 \mapsto 1$ and $c_2 \mapsto 1000$. Therefore, φ is satisfiable modulo `TB`.

4.4 Solver Architecture

The techniques described in this section can be combined to produce a simple and effective solver for non-trivial benchmarks. Figure 1 shows the algorithm used in our prototype. The solver implements a form of *counter-example guided refinement* where a failed *model-checking* step suggests new instances for the universally quantified formula. This method is also a variation of *model-based quantifier instantiation* [11] based on templates. The procedure `SMT` is an SMT solver for the quantifier-free bit-vector and uninterpreted function theory (QF_UFBV in SMT-LIB [1]). The procedure $\texttt{HeuristicInst}(\phi[\overline{x}])$ creates an initial set of ground instances of $\phi[\overline{x}]$ using heuristic instantiation and conjoins them with existing ground

solver(φ, TB)
  φ := Simplify(φ)
  w.l.o.g. assume φ is of the form $\psi \wedge \forall \overline{x}.\ \phi[\overline{x}]$
  $\rho := \psi \wedge$ HeuristicInst($\phi[\overline{x}]$)
  loop
    if SMT(ρ) = *unsat* **return** *unsat*
    M := TMF(ρ, TB)
    if $M = \bot$ **return** *unsat modulo* TB
    V := MC(φ, M)
    if $V = \top$ **return** (*sat*, M)
    $\rho := \rho \wedge \bigwedge_{\overline{v} \in V} \phi[\overline{v}]$

Fig. 1 QBVF solving algorithm.

instances ψ in the problem to form the initial set of instantiations ρ. Note that, during execution, ρ monotonically increases in size, so the procedures SMT and TMF can exploit incremental solving features available in state-of-the-art SMT solvers.

Theorem 2 *The algorithm in Figure 1 is complete modulo the given template* TB.

Proof The formula ρ increases monotonically. The conjunct added in every iteration is an instance of ϕ with all universals replaced by values from the counter-example V, thereby adding new quantifier instances to ρ in every iteration. Since the number of possible instantiations is finite, the process terminates. In case it does so with *unsat modulo* TB, there is no instance of the template TB that satisfies ρ. Since ρ is a conjunction of instances of ϕ, there is no model for ϕ modulo TB. □

The algorithm in Figure 1 is complete for QBVF if TMF never fails, that is, M is never $\bot$. This can be accomplished using a template that simply covers *all* relevant functions: Let us assume w.l.o.g that every function in φ has only one argument and it is a bit-vector of size 2^n. Then, using the template

$$ite(x = c_1, a_1, \ldots, ite(x = c_{2^n-1}, a_{2^n-1}, a_{2^n}) \ldots)$$

guarantees that TMF will never fail, where $c_1, \ldots c_{2^n-1}, a_1, \ldots, a_{2^n}$ are the template parameters. Of course, it is impractical to use this template in practice.

4.5 Symbolic quantifier instantiation

In many cases, using actual tuples of bit-vector values is not the best strategy for instantiating quantifiers. Instead, ground terms may be used to decide the formula more quickly.

Example 8 Let f be a function from bit-vectors of size 32 to bit-vectors of the same size in

$$(\forall x.\ f(x) \geq 1)\ \wedge\ f(a) < 1\ .$$

This formula is unsatisfiable and contains the ground term $\psi = f(a) < 1$. The algorithm in Figure 1 would require 2^{32} instantiations of x to decide this formula,

since it would enumerate bit-vector values as follows:

$$
\begin{aligned}
& f(0) \geq 1 \wedge f(a) < 1 \\
& f(0) \geq 1 \wedge f(1) \geq 1 \wedge f(a) < 1 \\
& \quad \ldots \\
& f(0) \geq 1 \wedge f(1) \geq 1 \wedge \ldots \wedge f(2^{32-2}) \geq 1 \wedge f(a) < 1 \\
& f(0) \geq 1 \wedge f(1) \geq 1 \wedge \ldots \wedge f(2^{32-2}) \geq 1 \wedge f(2^{32-1}) \geq 1 \wedge f(a) < 1 \,,
\end{aligned}
$$

always finding that the instance is satisfiable (for some a), except in the last instantiation, where the whole domain of f is fixed to function values greater or equal to 1. In the first iteration however, ρ contains ψ and therefore $f(a) < 1$. This ground term suggests a as a possibly helpful instantiation of x. We therefore obtain

$$f(a) \geq 1 \wedge f(a) < 1$$

which immediately establishes unsatisfiability.

The exact method that we use to obtain symbolic instantiations from ground instances is similar to the one used in [11]: Given a counter-example $(v_1, \ldots, v_n)$ in V, if there is a term t in ρ s.t. $M[\![t]\!] = v_i$, we use t instead of v_i to instantiate the quantifier. Of course, in practice, there may be several different terms to chose from. In this case we select the syntactically smallest one and break ties non-deterministically.

4.6 Additional Techniques for Solving QBVF

Templates may be used to eliminate uninterpreted function (predicate) symbols from any QBVF formula. The idea is to replace any function application $f_i(\overline{r})$ (ground or not) in a QBV formula φ with the template definition $t_i[\overline{r}, \overline{c}]$. The resultant formula φ' contains only uninterpreted constants and interpreted bit-vector operators. Therefore, *bit-blasting* can be used to encode φ' into QBF. This observation also suggests that template model finding is essentially approximating a NEXPTIME-complete problem (QBVF satisfiability) as a PSPACE-complete one (QBF satisfiability). Of course, the reduction is effective iff the size of the templates are polynomially bounded by the input formula size.

If the QBV formula is a conjunction of many universally quantified formulas, a more attractive approach is quantifier elimination using BDDs [3] or resolution and expansion [4]. Each universally quantified clause can be independently processed and the resultant formulas or clauses are combined. Another possibility is to apply this approach only to a selected subset of the universally quantified sub-formulas, and rely on the approach described in section 4.4 for the remaining ones.

Finally, first-order resolution and subsumption can also be used to derive new implied QBV universally quantified clauses and to delete redundant ones.

5 Experimental Results

To assess the efficacy of our method we present an evaluation of the performance of our QBVF solver which is integrated into version 3.0 of the Z3 SMT solver [24].

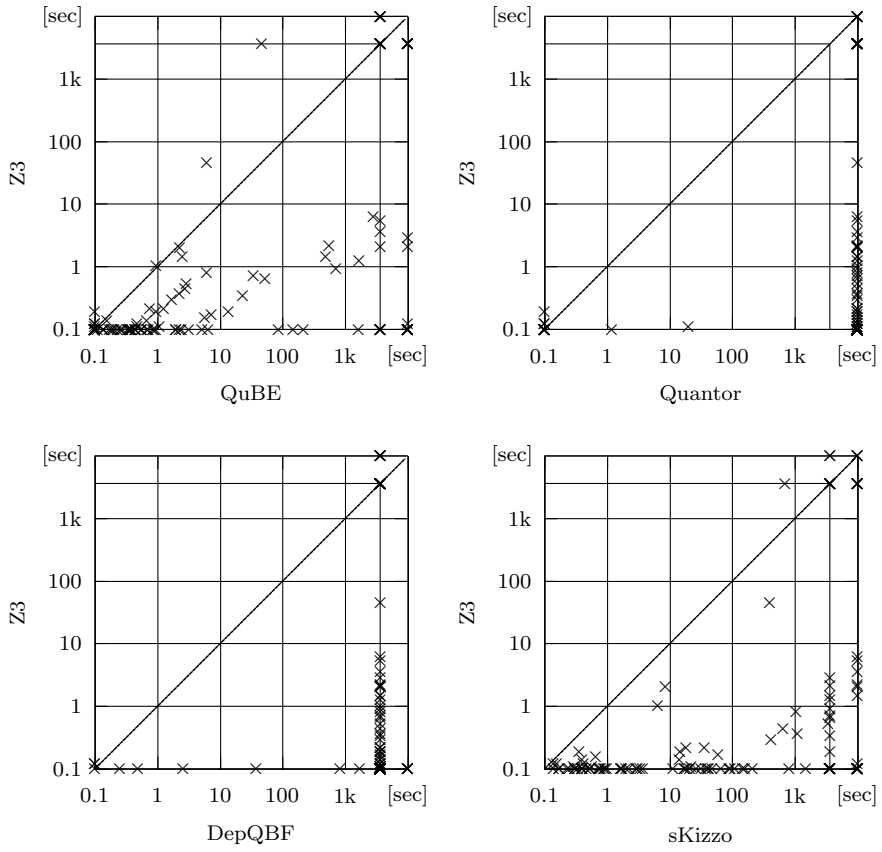

Fig. 2 Comparison of runtime on hardware fixpoint problems.

Our implementation first applies the simplifications described in section 4.1. It then iterates model checking and model finding as described in sections 4.2-4.5. The benchmarks that we use for our performance comparison are derived from two sources: a) hardware fixpoint checks and b) software ranking function synthesis [8]. It is not trivial to compare our QBVF solver with other systems, since most SMT solvers either lack support for or do not perform well in benchmarks containing bit-vectors and quantifiers.[3] In the past, QBF solvers have been used to attack these problems. We therefore compare to the state-of-the-art QBF solvers DepQBF 0.1 [21], Quantor 3.0 [4], QuBE 7.2 [12], and sKizzo 0.8.2 [3].

Formulas in the first set exhibit the structure of fixpoint formulas described in section 3. The circuits that we use as benchmarks are derived from a previous evaluation of VCEGAR [16][4] and were extracted using a customised version of

[3] Cf., e.g., the results of previous SMT-Competitions at `http://www.smt-comp.org/`

[4] These benchmarks are available at `http://www.cprover.org/hardware/`

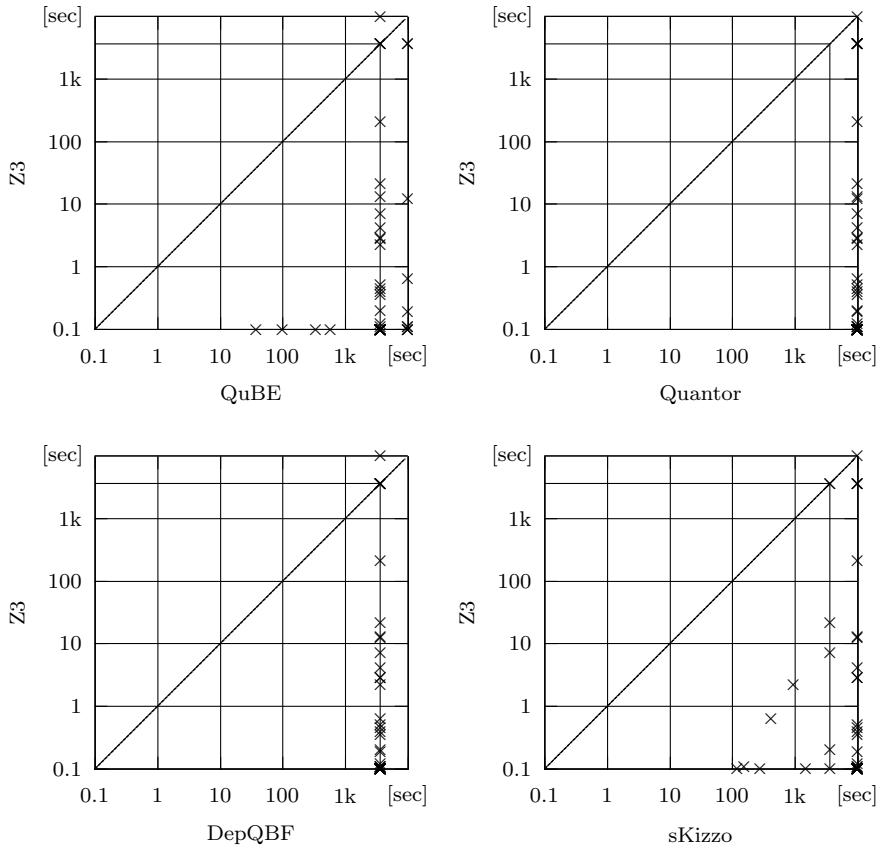

Fig. 3 Comparison of runtime on ranking function synthesis problems.

the EBMC bounded Model Checker[5], which is able to produce fixpoint checks in QBVF and QBF form. In total, this benchmark set contains 131 files.

Our second set of benchmarks cannot be directly encoded in QBF because they contain uninterpreted function symbols. Therefore, we decided to consider only ranking functions that are linear polynomials. By applying this template we can convert the problem to QBF as described in section 4.6. Thus, the problem here is to synthesise the coefficients for the polynomial. This benchmark set consists of 60 instances. Further details, e.g., on the size of the coefficients, were described in previous work [8].

All our benchmarks were extracted in two forms: in QBVF form (using SMT-LIB format) and in QBF form (in QDIMACS format) and they were executed on a Windows HPC cluster of Intel Xeon 2.5 GHz machines with a time limit of 3600 seconds and a memory limit of 2 GB. Data points at the 10,000 second line in Figures 2, 3, and 5 indicate that a solver exceeded the memory limit.

[5] EBMC is available at `http://www.cprover.org/ebmc/`

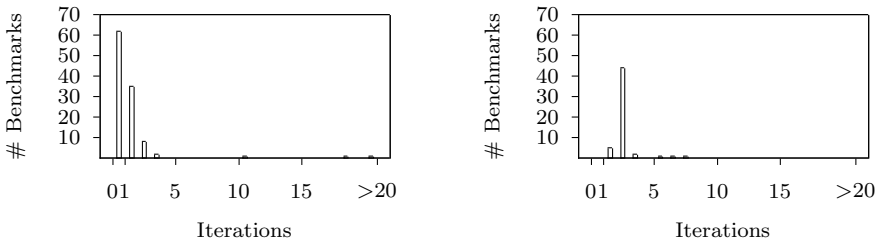

Fig. 4 The distribution of refinement iterations required on hardware fixpoint problems (left) and ranking function synthesis problems (right).

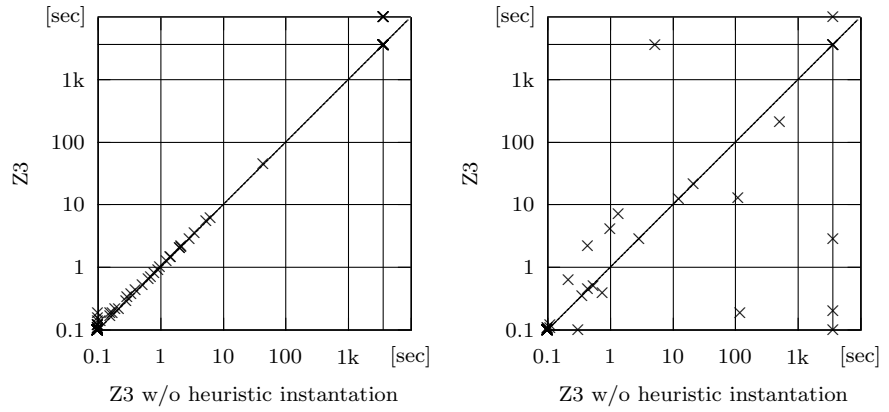

Fig. 5 Comparison of runtime of Z3 with and without heuristic instantiation on hardware fixpoint problems (left) and ranking function synthesis problems (right).

As demonstrated by Figure 2, Z3 outperforms all QBF solvers on almost all hardware fixpoint problems, outperforming all solvers by up to five orders of magnitude in runtime. It solves almost all instances in the this set (110 out of 131). Most of the benchmarks solved in this category (89 out of 110) are solved by our simplifications and rewriting rules alone. In the remaining cases, the model refinement algorithm requires less than 4 iterations, except for three benchmarks. On ranking function synthesis problems, the number of iterations required to find a model or to prove non-existence of a model is again very small: almost all instances require only one or two iterations and the maximum number of iterations is 7. In this benchmark set, Z3 solves 54 out of 60 benchmarks, outperforming all QBF solvers on all instances, as shown in Figure 3. Even though our algorithm exhibits similar speed-ups on both benchmark sets, the behaviour on the second set is quite different: As evident from Figure 4, none of the instances in this set is completely solved by the simplifications or rewriting rules; the model finding algorithm is required on each of them.

In addition to the comparative study of the solvers discussed so far, it is interesting to see what the impact of the initial heuristic quantifier instantiation on our algorithm is. Figure 5 compares Z3 to itself without heuristic quantifier instantia-

tion. It shows that most hardware fixpoint problems are unaffected by this, which is not surprising, because most of them are solved by our simplification rules alone. On the ranking function synthesis problems however, heuristic quantifier instantiation can still be very effective, being solely responsible for large speed-ups. On some problems however, it incurs a non-negligible performance penalty.

5.1 The impact of preprocessing

To assess the impact of each of the simplification rules described in section 4.1, we performed another set of experiments. On the same benchmark sets as before, we investigate by how much the runtime of the solver is degraded by disabling one simplification rule at a time. Figure 6 contains scatter plots, each of them comparing Z3 to a version of itself with the simplification rule indicated along the x-axis disabled. It is clear from these plots that all simplification rules have, on average, a positive effect on the runtime of the solver. Clearly, DER has the biggest impact and is solely responsible for speed-ups of up to three orders of magnitude in runtime on many hardware fixpoint benchmarks. Note that in some cases, disabling a simplification can have a positive effect. This is due to the fact that the search procedures employed for model finding receive a different formula and consequently may find different models; of course this is unintended but also unavoidable.

6 Related Work

In practice it is often the case that uninterpreted functions are not strictly required. In this case, QBVFs can be flattened into either a propositional formula or a quantified Boolean formula (QBF). This is possible because bit-vector variables may be treated as a vector of Boolean variables. Operations on bit-vectors may be bit-blasted, but this approach increases the size of the formula considerably (e.g., quadratically for multipliers), and structural information is lost. In case of quantified formulas, universal quantifiers can be expanded since each is a quantification over a finite domain of values. This usually results in an exponential increase of the formula size and is therefore infeasible in practice. An alternative method is to flatten the QBV formula without expanding the quantifiers. This results in a QBF and off-the-shelf decision procedures (QBF solvers, e.g., DepQBF [21], Quantor [4], QuBE [12], or sKizzo [3]) may be employed to decide the formula. In practice, the performance of QBF solvers has proven to be problematic, however. One of the potential issues resulting in bad performance may be the prenex clausal form of QBFs. It has thus been proposed to use non-prenex non-clausal form [9, 13], which was shown to be beneficial on certain types of formulas. However, all known decision procedures of this type fail to exploit any form of word-level information. A further problem with QBF solvers is that only few of them support certification, especially the construction of models for satisfiable instances. This is an absolute necessity for solvers employed in a synthesis context.

SMT QF_BV solvers. For some time now, SMT solvers for the quantifier-free fragment of bit-vector logic existed. Usually, these solvers are based on a small

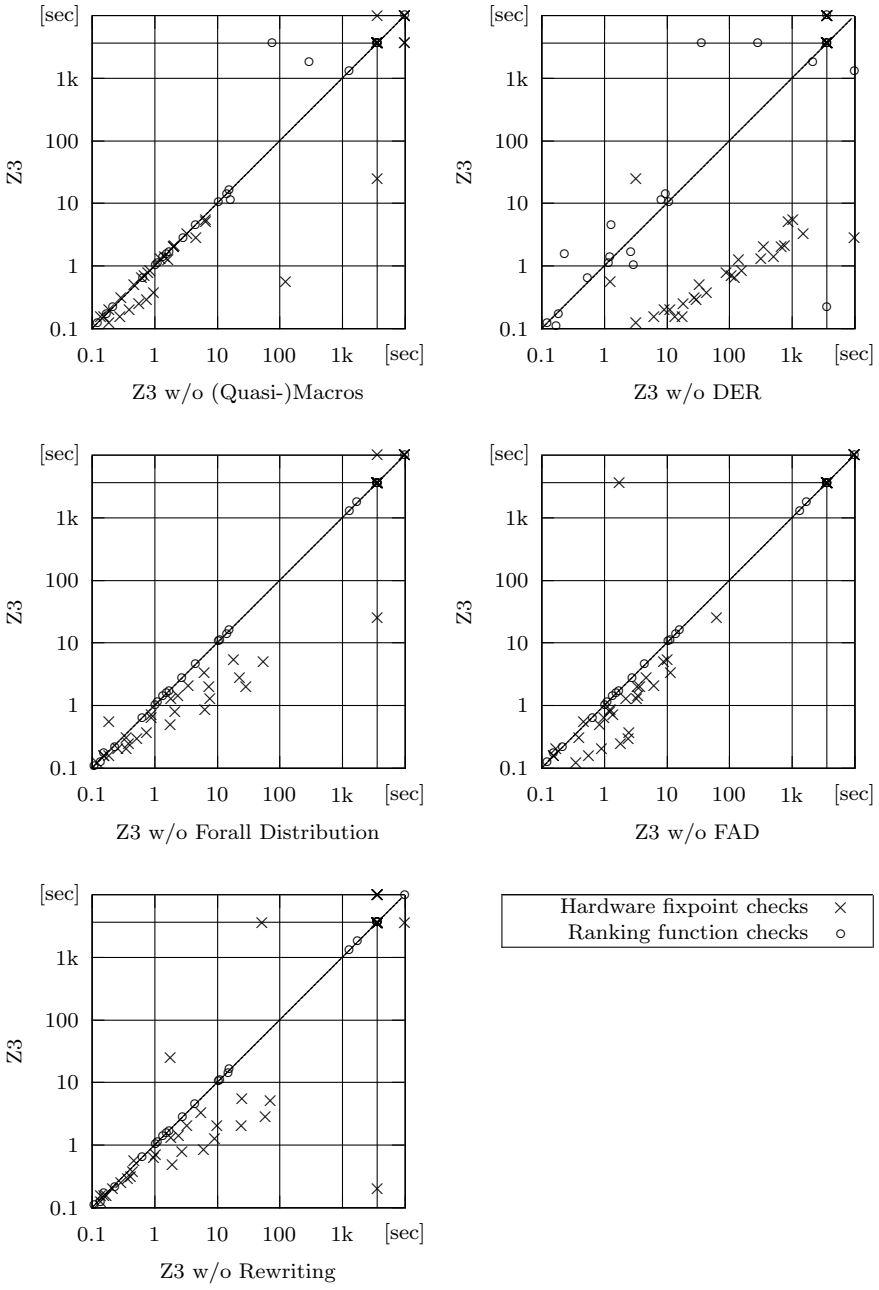

Fig. 6 The impact of the various simplifications on the solving time.

set of word-level simplifications and subsequent flattening (bit-blasting) to propositional formulas. Some solvers (e.g., SWORD [32]), try to incorporate word-level information while solving the flattened formula. Some tools also have limited support for quantifiers (e.g. BAT [22]), but this is usually restricted to either a single quantifier or a single alternation of quantifiers which may be expanded at feasible cost. Most SMT QF_BV solvers support heuristic instantiation of quantifiers based on E-matching [23]. On some unsatisfiable instances, this may terminate with a conclusive result, but it is of course not a solution to the general problem. The method that we propose uses SMT solvers for the quantifier-free fragment to decide intermediate formulas and therefore represents an extension of SMT techniques to the more general QBV logic.

Synthesis tools. Finally, there is recent and active interest in using modern SMT solvers in the context of synthesis of inductive loop invariants [28], synthesis via sketching [27] and synthesis of program fragments [17], such as sorting, matrix multiplication, de-compression, graph, and bit-manipulating algorithms. These applications share a common trait in the way they use their underlying symbolic solver. They search a template *vocabulary* of instructions, that are composed as a model in a satisfying assignment. This approach was the main inspiration for the template based model finder described in section 4.3.

7 Conclusion

Quantified bit-vector (QBV) logic is ideally suited as an interface between verification or synthesis tools and underlying decision procedures. Decision procedures for different fragments of this logic are required in virtually every verification or synthesis technique, making the QBV logic one of the most practically relevant logics. We present a new approach to solving quantified bit-vector formulas based on a set of simplifications and rewrite rules, as well as a new model finding algorithm based on an iterative refinement scheme. Through an evaluation on benchmarks that stem from hardware and software applications, we are able to demonstrate that our approach is up to five orders of magnitude faster when compared to the popular approach of flattening the formula to QBF.

Acknowledgments. We would like to thank the anonymous reviewers of this paper for their thorough reviewing, which helped greatly improve the quality of this paper. Their comments on theoretical mistakes as well as presentational issues were an invaluable aid in improving this paper.

References

1. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2010)
2. Barrett, C., Tinelli, C.: CVC3. In: Proc. of CAV, no. 4590 in LNCS. Springer (2007)
3. Benedetti, M.: Evaluating QBFs via Symbolic Skolemization. In: Proc. of LPAR, no. 3452 in LNCS. Springer (2005)
4. Biere, A.: Resolve and expand. In: Proc. of SAT, no. 3542 in LNCS. Springer (2005)
5. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Proc. of TACAS, no. 5505 in LNCS. Springer (2009)

6. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT solver. In: Proc. of CAV, no. 5123 in LNCS. Springer (2008)
7. Colón, M.: Schema-guided synthesis of imperative programs by constraint solving. In: Proc. of Intl. Symp. on Logic Based Program Synthesis and Transformation, no. 3573 in LNCS. Springer (2005)
8. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking function synthesis for bit-vector relations. In: Proc. of TACAS, no. 6015 in LNCS. Springer (2010)
9. Egly, U., Seidl, M., Woltran, S.: A solver for QBFs in negation normal form. Constraints **14**(1) (2009)
10. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Proc. of CAV, no. 4590 in LNCS. Springer (2007)
11. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Proc. of CAV, no. 5643 in LNCS. Springer (2009)
12. Giunchiglia, E., Narizzano, M., Tacchella, A.: QuBE++: An efficient QBF solver. In: Proc. of FMCAD, no. 3312 in LNCS. Springer (2004)
13. Goultiaeva, A., Iverson, V., Bacchus, F.: Beyond CNF: A circuit-based QBF solver. In: Proc. of SAT, no. 5584 in LNCS. Springer (2009)
14. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: Proc. of VMCAI, no. 5403 in LNCS. Springer (2009)
15. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
16. Jain, H., Kroening, D., Sharygina, N., Clarke, E.M.: Word-level predicate-abstraction and refinement techniques for verifying RTL verilog. IEEE Trans. on CAD of Int. Circuits and Systems **27**(2) (2008)
17. Jha, S., Gulwani, S., Seshia, S., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proc. of ICSE. ACM (2010)
18. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: Proc. of FMCAD. IEEE (2006)
19. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebra. In: Proc. Conf. on Computational Problems in Abstract Algebra. Pergamon Press (1970)
20. Lewis, H.R.: Complexity results for classes of quantificational formulas. J. Comput. Syst. Sci. **21**(3) (1980)
21. Lonsing, F., Biere, A.: Integrating dependency schemes in search-based QBF solvers. In: Proc. of SAT, no. 6175 in LNCS. Springer (2010)
22. Manolios, P., Srinivasan, S.K., Vroon, D.: BAT: The bit-level analysis tool. In: Proc. of CAV, no. 4590 in LNCS. Springer (2007)
23. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Proc. of CADE, no. 4603 in LNCS. Springer (2007)
24. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of TACAS, no. 4963 in LNCS. Springer (2008)
25. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. of POPL. ACM (1989)
26. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Proc. of VMCAI, no. 2937 in LNCS. Springer (2004)
27. Solar-Lezama, A., Jones, C.G., Bodík, R.: Sketching concurrent data structures. In: Proc. of PLDI. ACM (2008)
28. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: Proc. of PLDI. ACM (2009)
29. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: Proc. of POPL. ACM (2010)
30. Staber, S., Bloem, R.: Fault localization and correction with QBF. In: Proc. of SAT, no. 4501 in LNCS. Springer (2007)
31. Turing, A.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines (1949)
32. Wille, R., Fey, G., Große, D., Eggersglüß, S., Drechsler, R.: Sword: A SAT like prover using word level information. In: Proc. Intl. Conf. on VLSI of System-on-Chip. IEEE (2007)