Efficiently Solving Quantified Bit-Vector Formulas

Christoph M. Wintersteiger
Computer Systems Institute
ETH Zurich
Zurich, Switzerland
christoph.wintersteiger@inf.ethz.ch

Youssef Hamadi
Microsoft Research Cambridge
7 JJ Thomson Avenue
Cambridge CB3 0FB, UK
youssefh@microsoft.com

Leonardo de Moura
Microsoft Research Redmond
One Microsoft Way
Redmond, WA, 98074, USA
leonardo@microsoft.com

***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.**

I. 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 are a requirement and oftentimes the crucial ingredient that defines the efficency 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 [23]) 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 [23], [20], while for software this can take different shapes: inferring program invariants [16], finding ranking functions for termination analysis [28], [24], [8], program fragment synthesis [26], or constructing bugfixes following an error-description [27] 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 evalutate 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.

II. Background

We will assume the usual notions and terminology of first order logic and model theory. 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. 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 will always omit the subscript. 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. 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 will 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 and y 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 $f\colon n_1, \ldots, n_k \rightarrow n_r$ to denote to denote that function symbol f has arity k, argument bit-vectors have sizes n_1, ..., 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, ..., x_n, and similarily $t[x_1, \ldots, x_n]$ 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 $\{F\} \cup BitVec$.

III. 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 (existental) 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.

Example 1: Arrays can be easily encoded in QBVF using quantifiers and uninterpreted function symbols. In the following formula, the uninterpreted functions f and f' are used to represent arrays from bit-vectors of size 8 to bit-vectors of the same size, and f' is essentially the array f updated at position $a+1$ with value 0:

$$f'(a+1) = 0 \ \wedge \ (\forall x : 8.\ x = a+1 \vee f'(x) = f(x)) \ .$$

Quantified *Boolean* formulas (QBF) are a generalization 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. The EPR fragment of 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 NEXPTIME-complete.

Theorem 1: The satisfiability problem for QBVF is NEXPTIME-complete.[1]

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 synthesing code [26], loop invariants [7], [16] and ranking functions [8] for finite-state programs. 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.

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

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

[1]For a proof of this theorem, see Appendix A.

existentially quantified variable, as it is impossible to express the correct variable dependencies in a linear quantifier prefix.

IV. 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 QBVF 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).

A. Simplifications & Rewriting

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

1) Miniscoping: Miniscoping is a well-known technique for minimizing the scope of quantifiers [17]. We apply it after converting the formula to negation normal form. The basic idea is to distribute universal (existential) quantifiers over conjunctions (disjunctions). This transformation is particularly important in our context because it increases the applicability of rules based on rewriting and macros. We may also limit the scope of a quantifier if a sub-formula does not contain the quantified variable. That is,

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

when G does not contain x. We use a similar rule for existential quantifiers over disjunctions.

2) Skolemization: Similarly to first-order theorem provers, in our solver, existentially quantified variables are eliminated using *Skolemization*. 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.

3) A conjunction of universally quantified formulas: After NNF conversion, 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.

4) Destructive Equality Resolution (DER): DER allows us to solve a negative equality literal by simply applying the following 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 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 following 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 when visiting node x_i, we remove x_i from G and move $x_i \neq t_i$ 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 [12].

5) Rewriting: The idea of using rewriting for performing equational reasoning is not new. It traces back to the work developed in the context of Knuth-Bendix completion [21]. 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 $f(x,a) \rightsquigarrow x$. Thus, 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. So, we are not interested in computing critical pairs and 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., [17]). We found that any term ordering, where interpreted symbols (e.g., +, *) are considered "small", works 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. 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 and reduce it to $a+1$.

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 QBVF 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 IV-B.

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. Now, the quasi-macro was transformed into a macro, the quantifier can be eliminated using macro expansion.

Example 2 (Quasi-Macro): $\forall x.f(x+1, x-1) = x$ 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, i n many benchmarks we found quasi-macros that are bigger versions of

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

7) Function Argument Discrimination (FAD): We have observed that after applying DER the i-th argument of many function applications is always a 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 $f(v', \overline{t})$ to $f_v(\overline{t})$ when $v = v'$, and $f'(v', \overline{t})$ otherwise. The transformation can be applied again to the f' applications if their first argument is again a bit-vector value.

Example 3 (FAD): 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 $f_{0,1}$.

8) Other simplifications: As many other SMT solvers for bit-vector theory ([6], [5], [2]), our QBVF solver implements several bit-vector specific rewriting/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.

B. 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. With a small 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) $f\colon n_1, \ldots, n_k \to n_r$ 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 4 (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

$$M_a := \{f(x) \mapsto ite(x \geq 0, x - 1, x + 3),\ a \mapsto -1,\ b \mapsto 1\}$$

is a model for φ_a. For instance, we have $M[\![f(a)]\!] = 2$.
Usually, SMT solvers represent the interpretation of unintepreted 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 $\{0 \mapsto 1,\ 2 \mapsto 3,\ else \mapsto 4\}$ 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 $g(x) \mapsto ite(x = 0, 1, ite(x = 2, 3, 4))$. Our approach for enconding 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 $f(x) \mapsto ite(x \geq 0, x - 1, x + 3)$ 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 [13]. 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 5: For instance, in Example 4, the structure M_a satisfies $\forall x.\ \neg(x \geq 0) \vee f(x) < x$ because

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

is unsatisfiable. Let M_b be a structure identical to M_a in Example 4, 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 $\overline{v}$ such that $M\{\overline{x} \mapsto \overline{v}\}[\![\varphi[\overline{x}]]\!]$ evaluates to $false$.

C. Template Based Model Finding

In principle, the verification and synthesis problems described in section III 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 techniques based on *heuristic-quantifier instantiation* [9].

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, when searching for a ranking function, the synthesis tool may limit the search to functions that are linear combinations of the input. 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 just 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 s.t. the interpretation of f is $t[\overline{x}, \overline{v}]$ for some bit-vector tuple $\overline{v}$ if such a structure exists. TMF returns $\bot$ 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 6 (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.

D. 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* [13] 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 instantes of $\phi[\overline{x}]$ using heuristic instantiation. Note that the formula ρ is monotonically increasing 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.[2]

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

[2]For a proof of this theorem, see Appendix B.

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

Fig. 1. QBVF solving algorithm.

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. Therefore, in our implementation, we consider templates of increasing complexity. We essentially use an outer-loop that automatically increases the size of the templates whenever the inner-loop returns *unsat modulo* TB.

In many cases, using actual tuples of bit-vector values is not the best strategy for instantiating quantifers. For example, assume f is a function from bit-vectors of size 32 to bit-vectors of the same size in

$$(\forall x.\ f(x) \geq 0), \quad f(a) < 0\ .$$

To prove this formula to be unsatisfiable, we should instantiate the quantifier with a instead of the 2^{32} possible bit-vector values. Therefore, we use an approach similar to the one used in [13]. Given a tuple $(v_1, \ldots, v_n)$ 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, we may have several different t's to chose from. In this case we select the syntactically smallest one, and break ties non-deterministically.

E. 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].

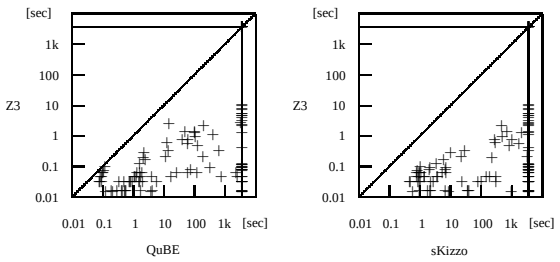

Fig. 2. Hardware fixpoint checks: QuBE & sKizzo vs. Z3

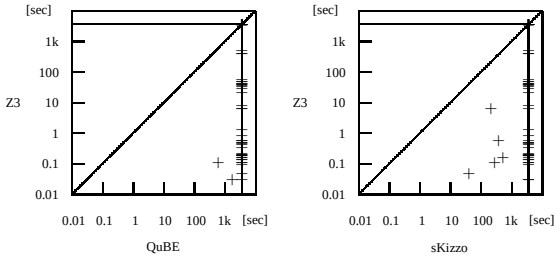

Fig. 3. Ranking function synthesis: QuBE & sKizzo vs. Z3

Each universally quantified clause can be independently processed and the resultant formulas/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 IV-D 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.

V. Experimental Results

To assess the efficacy of our method we present an evaluation of the performance of a preliminary QBVF solver based on the code-base of the Z3 SMT solver [10]. Our prototype first applies the simplifications described in section IV-A. It then iterates model checking and model finding as described in sections IV-B and IV-C. 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 do not perform well in benchmarks containing bit-vectors and quantifiers. In the past, QBF solvers have been used to attack these problems. We therefore compare to the state-of-the-art QBF solvers sKizzo [3] and QuBE [14].

Formulas in the first set exhibit the structure of fixpoint formulas described in section III. The circuits that we use as benchmarks are derived from a previous evaluation of VCEGAR [18][3] and were extracted using a customized version of the EBMC bounded Model Checker[4], 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. So, 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 IV-E. Thus, the problem here is to synthesise the coefficients for the polynomial. Further details, especially on the size of the coefficients, were described previously [8].

All our benchmarks were extracted in two forms: in QBVF form (using SMT-LIB format) and in QBF form (using the QDIMACS format) and they were executed on a Windows HPC cluster of AMD Athlon 2 GHz machines with a time limit of 3600 seconds and a memory limit of 2 GB.

As indicated by Figure 2 our approach outperforms the QBF solvers on all instances, sometimes by up to five orders of magnitude and it solves almost all instances in the benchmark set (110 out of 131). Most of the benchmarks solved in this category (87 out of 110) are solved by our simplifications and rewriting rules only. In the remaining cases, the model refinement algorithm takes less then 10 iterations.

Figure 3 show the results for the ranking function benchmark set. Again, our algorithm outperforms the QBF solvers by up to five orders of magnitude. The number of iterations required to find a model or prove non-existence of a model in these benchmarks is again very small: almost all instances require only one or two iterations and the maximum number of iterations is 9. Even though our algorithm exhibits similar speedups on both benchmark sets, the behaviour on the second set is quite different: 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.[5]

VI. 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) like sKizzo [3], Quantor [4] or QuBE [14] may be employed to decide the formula. In practice, the performance of QBF solvers has proven to be problematic, however.

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

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

[5]More experimental data is provided in Appendix C.

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 [11], [15]. This has been demonstrated to be beneficial on certain types of formulas, but all known decision procedures 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, those solvers are based on a small set of word-level simplifications and subsequent flattening (bit-blasting) to propositional formulas. Some solvers (e.g., SWORD [29]), 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 [9]. 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 [25] and synthesis of program fragments [19], such as sorting, matrix multiplication, decompression, 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 IV-C.

VII. Conclusion

Quantified bit-vector logic (QBV) 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 QBV 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 a popular approach of flattening the formula to QBF.

References

[1] C. Barrett, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2010.

[2] C. Barrett and C. Tinelli, "CVC3," in *Proc. of CAV*, ser. LNCS, no. 4590. Springer, 2007.

[3] M. Benedetti, "Evaluating QBFs via Symbolic Skolemization," in *Proc. of LPAR*, ser. LNCS, no. 3452. Springer, 2005.

[4] A. Biere, "Resolve and expand," in *Proc. of SAT'04 (Revised Selected Papers)*, ser. LNCS, no. 3542. Springer, 2005.

[5] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Proc. of TACAS*, ser. LNCS, no. 5505. Springer, 2009.

[6] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4 SMT solver," in *CAV*, ser. LNCS, no. 5123. Springer, 2008.

[7] M. Colón, "Schema-guided synthesis of imperative programs by constraint solving," in *Proc. of Intl. Symp. on Logic Based Program Synthesis and Transformation*, ser. LNCS, no. 3573. Springer, 2005.

[8] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger, "Ranking function synthesis for bit-vector relations," in *Proc. of TACAS*, ser. LNCS, no. 6015. Springer, 2010.

[9] L. de Moura and N. Bjørner, "Efficient e-matching for SMT solvers," in *Proc. of CADE*, ser. LNCS, no. 4603. Springer, 2007.

[10] ——, "Z3: An efficient SMT solver," in *Proc. of TACAS*, ser. LNCS, no. 4963. Springer, 2008.

[11] U. Egly, M. Seidl, and S. Woltran, "A solver for QBFs in negation normal form," *Constraints*, vol. 14, no. 1, 2009.

[12] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proc. of CAV*, ser. LNCS, no. 4590. Springer, 2007.

[13] Y. Ge and L. de Moura, "Complete instantiation for quantified formulas in satisfiabiliby modulo theories," in *CAV*, ser. LNCS, no. 5643. Springer, 2009.

[14] E. Giunchiglia, M. Narizzano, and A. Tacchella, "QuBE++: An efficient QBF solver," in *Proc. of FMCAD*, ser. LNCS, no. 3312. Springer, 2004.

[15] A. Goultiaeva, V. Iverson, and F. Bacchus, "Beyond CNF: A circuit-based QBF solver," in *Proc. of SAT*, ser. LNCS, no. 5584. Springer, 2009.

[16] S. Gulwani, S. Srivastava, and R. Venkatesan, "Constraint-based invariant inference over predicate abstraction," in *Proc. of VMCAI*, ser. LNCS, no. 5403. Springer, 2009.

[17] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[18] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word-level predicate-abstraction and refinement techniques for verifying rtl verilog," *IEEE Trans. on CAD of Int. Circuits and Systems*, vol. 27, no. 2, 2008.

[19] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proc. of ICSE*. ACM, 2010.

[20] B. Jobstmann and R. Bloem, "Optimizations for LTL synthesis," in *FMCAD*. IEEE, 2006.

[21] D. E. Knuth and P. B. Bendix, "Simple word problems in universal algebra." in *Proc. Conf. on Computational Problems in Abstract Algebra*. Pergamon Press, 1970.

[22] P. Manolios, S. K. Srinivasan, and D. Vroon, "BAT: The bit-level analysis tool," in *CAV*, ser. LNCS, no. 4590. Springer, 2007.

[23] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. of POPL*. ACM, 1989.

[24] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Proc. of VMCAI*, ser. LNCS, no. 2937. Springer, 2004.

[25] S. Srivastava and S. Gulwani, "Program verification using templates over predicate abstraction," in *Proc. of PLDI*. ACM, 2009.

[26] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *Proc. of POPL*. ACM, 2010.

[27] S. Staber and R. Bloem, "Fault localization and correction with QBF," in *SAT*, ser. LNCS, no. 4501. Springer, 2007.

[28] A. Turing, "Checking a large routine," in *Report of a Conference on High Speed Automatic Calculating Machines*, 1949.

[29] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler, "Sword: A SAT like prover using word level information," in *Proc. Intl. Conf. on Very Large Scale Integration of System-on-Chip*. IEEE, 2007.

APPENDIX

A. Proof of Theorem 1

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

1) QBVF ⇒ EPR: Given a QBV formula φ, w.l.o.g. we assume φ is in CNF. The first step is to flat 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 into $\forall \overline{x}, y.\ y \neq t \vee C[y]$. 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

$$\forall x_1, x_2, y_1, y_2, y_3.\ y_1 \neq g(x_2) \vee y_2 \neq f(x_1, y_1) \vee y_3 \neq g(x_1) \vee y_2 \leq y_3$$

Next, for each uninterpreted function f where the range 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 $x \neq f(y)$ is encoded as

$$\begin{array}{l}((x_1 = \top)\ xor\ p_{f_1}(y_1, \ldots, y_n)) \vee \\ \ldots \\ ((x_m = \top)\ xor\ p_{f_m}(y_1, \ldots, y_n))\end{array}$$

Other atoms are encoded in a similar way. 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, in the following QBV formula, assume all sorts are bit-vectors of size 2.

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

After flattening, we have:

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

Then, after bit-blasting, we have:

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

2) EPR ⇒ 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. Therefore, in principle, it should be straightforward to reduce a EPR formula to QBVF. 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 QBVF 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.

B. Proof of Theorem 2

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 must terminate. In case it terminates 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.

C. Experimental results in detail

Figures 4, 5, 6 and 7 are bigger versions of the Figures 2 and 3. Tables I and II provide all the runtimes (in seconds) and results of our experiments. They also include the runtimes for the QBF solver Quantor.

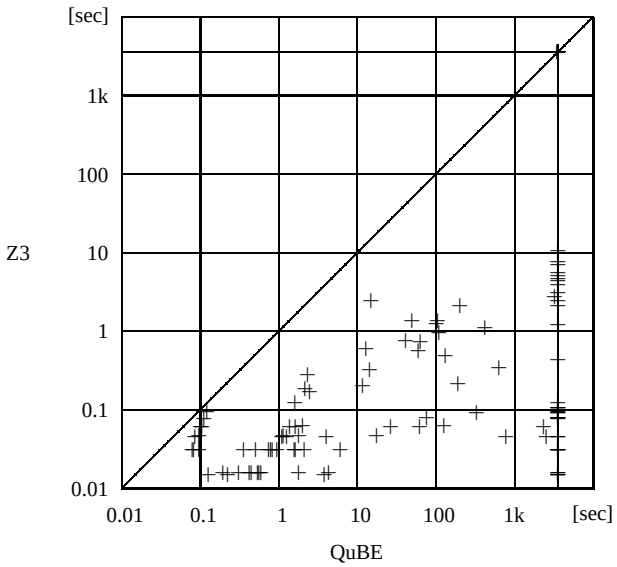

Fig. 4. Hardware fixpoint checks: QuBE vs. Z3

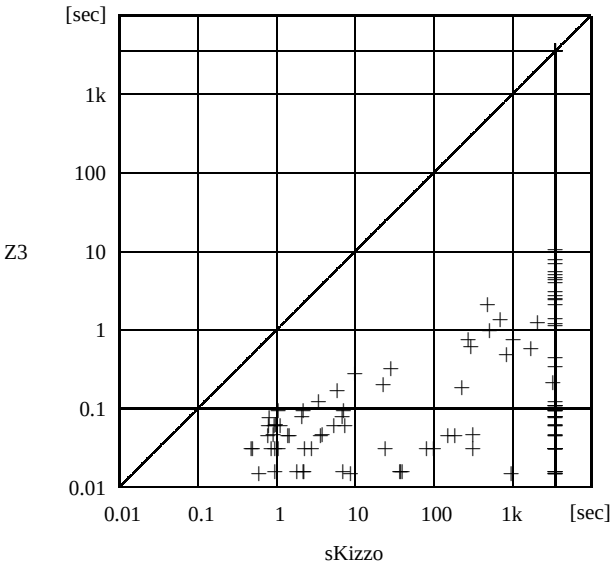

Fig. 5. Hardware fixpoint checks: sKizzo vs. Z3

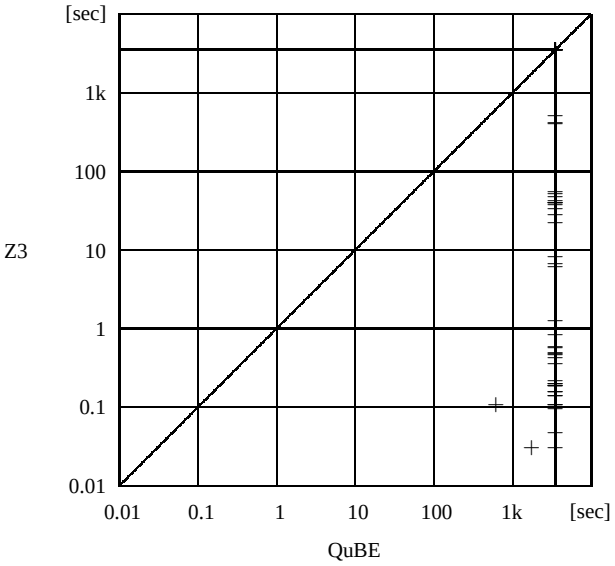

Fig. 6. Ranking function synthesis: QuBE vs. Z3

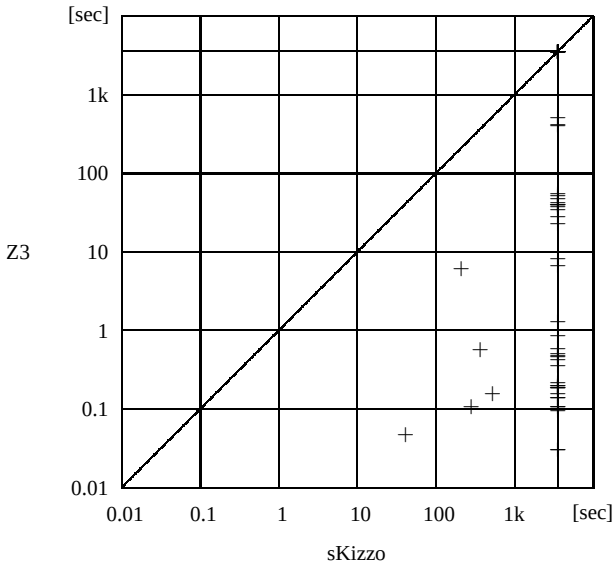

Fig. 7. Ranking function synthesis: sKizzo vs. Z3

	sKizzo	QuBE	Quantor	Z3	Result
AR-fixpoint-1.qdimacs	TIME	MEM	MEM	0.077	unsat
AR-fixpoint-10.qdimacs	MEM	MEM	TIME	0.124	unsat
AR-fixpoint-2.qdimacs	TIME	MEM	MEM	0.078	unsat
AR-fixpoint-3.qdimacs	TIME	MEM	TIME	0.078	unsat
AR-fixpoint-4.qdimacs	MEM	MEM	TIME	0.078	unsat
AR-fixpoint-5.qdimacs	MEM	MEM	TIME	0.094	unsat
AR-fixpoint-6.qdimacs	MEM	MEM	TIME	0.109	unsat
AR-fixpoint-7.qdimacs	MEM	MEM	TIME	0.094	unsat
AR-fixpoint-8.qdimacs	MEM	MEM	TIME	0.109	unsat
AR-fixpoint-9.qdimacs	MEM	MEM	TIME	0.109	unsat
cache-coherence-2-fixpoint-1.qdimacs	3298.794	193.22	MEM	0.218	unsat
cache-coherence-2-fixpoint-2.qdimacs	TIME	TIME	MEM	1.217	unsat
cache-coherence-2-fixpoint-3.qdimacs	TIME	TIME	MEM	2.417	unsat
cache-coherence-2-fixpoint-4.qdimacs	TIME	TIME	MEM	3.946	unsat
cache-coherence-2-fixpoint-5.qdimacs	MEM	TIME	MEM	7.098	unsat
cache-coherence-2-fixpoint-6.qdimacs	TIME	TIME	MEM	10.748	unsat
cache-coherence-3-fixpoint-1.qdimacs	TIME	630.714	MEM	0.343	unsat
cache-coherence-3-fixpoint-2.qdimacs	TIME	TIME	MEM	2.09	unsat
cache-coherence-3-fixpoint-3.qdimacs	TIME	TIME	MEM	4.461	unsat
ethernet-fixpoint-1.qdimacs	1036.26	63.214	MEM	0.748	unsat
ethernet-fixpoint-2.qdimacs	MEM	3266.96	MEM	2.793	unsat
ethernet-fixpoint-3.qdimacs	MEM	TIME	MEM	4.696	unsat
ethernet-fixpoint-4.qdimacs	TIME	TIME	MEM	9.999	unsat
itc-b13-fixpoint-1.qdimacs	2.277	1.643	MEM	0.031	unsat
itc-b13-fixpoint-10.qdimacs	704.89	105.746	MEM	1.357	sat
itc-b13-fixpoint-2.qdimacs	5.94	2.483	MEM	0.171	unsat
itc-b13-fixpoint-3.qdimacs	23.183	11.654	MEM	0.203	sat
itc-b13-fixpoint-4.qdimacs	29.02	14.42	MEM	0.328	sat
itc-b13-fixpoint-5.qdimacs	850.897	130.657	MEM	0.484	sat
itc-b13-fixpoint-6.qdimacs	1755.936	59.454	MEM	0.577	sat
itc-b13-fixpoint-7.qdimacs	277.154	41.524	MEM	0.764	sat
itc-b13-fixpoint-8.qdimacs	515.197	109.94	MEM	0.967	sat
itc-b13-fixpoint-9.qdimacs	TIME	417.43	MEM	1.123	sat
pi-bus-fixpoint-1.qdimacs	TIME	TIME	MEM	0.437	unsat
pi-bus-fixpoint-2.qdimacs	TIME	TIME	MEM	3.089	unsat
pi-bus-fixpoint-3.qdimacs	TIME	TIME	MEM	5.132	unsat
sdlx-fixpoint-1.qdimacs	3.487	1.61	MEM	0.124	unsat
sdlx-fixpoint-10.qdimacs	TIME	TIME	MEM	TIME	?
sdlx-fixpoint-2.qdimacs	10.17	2.32	MEM	0.281	unsat
sdlx-fixpoint-3.qdimacs	298.317	12.854	MEM	0.608	unsat
sdlx-fixpoint-4.qdimacs	2096.083	101.177	MEM	1.232	unsat
sdlx-fixpoint-5.qdimacs	490.48	202.53	MEM	2.121	unsat
sdlx-fixpoint-6.qdimacs	MEM	TIME	MEM	TIME	?
sdlx-fixpoint-7.qdimacs	TIME	TIME	MEM	TIME	?
sdlx-fixpoint-8.qdimacs	TIME	TIME	MEM	TIME	?
sdlx-fixpoint-9.qdimacs	TIME	TIME	MEM	TIME	?
small-bug1-fixpoint-1.qdimacs	0.507	0.957	0.17	0	sat
small-bug1-fixpoint-10.qdimacs	1.074	0.124	0.357	0.094	sat
small-bug1-fixpoint-2.qdimacs	0.48	0.08	0.147	0.031	sat
small-bug1-fixpoint-3.qdimacs	0.5	0.083	0.16	0.031	sat
small-bug1-fixpoint-4.qdimacs	0.787	0.087	0.163	0.046	sat
small-bug1-fixpoint-5.qdimacs	0.873	0.097	0.19	0.031	sat
small-bug1-fixpoint-6.qdimacs	0.924	0.097	0.184	0.047	sat
small-bug1-fixpoint-7.qdimacs	0.797	0.103	0.233	0.062	sat
small-bug1-fixpoint-8.qdimacs	0.96	0.107	0.21	0.062	sat
small-bug1-fixpoint-9.qdimacs	0.826	0.113	0.226	0.077	sat
small-dyn-partition-fixpoint-1.qdimacs	0.61	0.127	0.186	0.015	unsat
small-dyn-partition-fixpoint-10.qdimacs	2.206	TIME	MEM	0.093	unsat
small-dyn-partition-fixpoint-2.qdimacs	0.963	0.513	2.42	0.031	unsat
small-dyn-partition-fixpoint-3.qdimacs	0.97	4.383	4.25	0.016	unsat
small-dyn-partition-fixpoint-4.qdimacs	1.064	6.043	33.11	0.031	unsat
small-dyn-partition-fixpoint-5.qdimacs	0.983	125.81	MEM	0.063	unsat
small-dyn-partition-fixpoint-6.qdimacs	1.41	776.993	MEM	0.046	unsat
small-dyn-partition-fixpoint-7.qdimacs	1.123	2347.22	MEM	0.062	unsat
small-dyn-partition-fixpoint-8.qdimacs	1.454	2538.42	MEM	0.046	unsat
small-dyn-partition-fixpoint-9.qdimacs	2.14	TIME	MEM	0.078	unsat

	sKizzo	QuBE	Quantor	Z3	Result
small-equiv-fixpoint-1.qdimacs	MEM	TIME	MEM	0.015	sat
small-equiv-fixpoint-10.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-2.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-3.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-4.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-5.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-6.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-7.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-8.qdimacs	MEM	TIME	MEM	TIME	?
small-equiv-fixpoint-9.qdimacs	MEM	TIME	MEM	TIME	?
small-pipeline-fixpoint-1.qdimacs	TIME	TIME	MEM	0.016	unsat
small-pipeline-fixpoint-10.qdimacs	MEM	TIME	MEM	TIME	?
small-pipeline-fixpoint-2.qdimacs	TIME	TIME	MEM	0.031	unsat
small-pipeline-fixpoint-3.qdimacs	TIME	TIME	MEM	0.093	unsat
small-pipeline-fixpoint-4.qdimacs	TIME	TIME	MEM	TIME	?
small-pipeline-fixpoint-5.qdimacs	TIME	TIME	MEM	TIME	?
small-pipeline-fixpoint-6.qdimacs	TIME	TIME	MEM	TIME	?
small-pipeline-fixpoint-7.qdimacs	TIME	TIME	MEM	TIME	?
small-pipeline-fixpoint-8.qdimacs	TIME	TIME	MEM	TIME	?
small-pipeline-fixpoint-9.qdimacs	MEM	TIME	MEM	TIME	?
small-seq-fixpoint-1.qdimacs	976.613	3.84	MEM	0.015	unsat
small-seq-fixpoint-10.qdimacs	TIME	TIME	MEM	0.031	unsat
small-seq-fixpoint-2.qdimacs	MEM	TIME	MEM	0.015	unsat
small-seq-fixpoint-3.qdimacs	MEM	TIME	MEM	0.016	unsat
small-seq-fixpoint-4.qdimacs	TIME	TIME	MEM	0.015	unsat
small-seq-fixpoint-5.qdimacs	TIME	TIME	MEM	0.031	unsat
small-seq-fixpoint-6.qdimacs	TIME	TIME	MEM	0.031	unsat
small-seq-fixpoint-7.qdimacs	TIME	TIME	MEM	0.031	unsat
small-seq-fixpoint-8.qdimacs	TIME	TIME	MEM	0.046	unsat
small-seq-fixpoint-9.qdimacs	TIME	TIME	MEM	0.046	unsat
small-swap1-fixpoint-1.qdimacs	8.813	0.223	MEM	0.015	unsat
small-swap1-fixpoint-10.qdimacs	TIME	2.02	MEM	0.063	sat
small-swap1-fixpoint-2.qdimacs	24.506	0.36	MEM	0.031	sat
small-swap1-fixpoint-3.qdimacs	37.483	0.427	MEM	0.016	sat
small-swap1-fixpoint-4.qdimacs	TIME	0.6	MEM	0.016	sat
small-swap1-fixpoint-5.qdimacs	TIME	0.79	MEM	0.031	sat
small-swap1-fixpoint-6.qdimacs	MEM	0.947	MEM	0.031	sat
small-swap1-fixpoint-7.qdimacs	TIME	1.157	MEM	0.047	sat
small-swap1-fixpoint-8.qdimacs	TIME	1.383	MEM	0.062	sat
small-swap1-fixpoint-9.qdimacs	TIME	1.626	MEM	0.062	sat
small-swap2-fixpoint-1.qdimacs	0.55	0.163	MEM	0	unsat
small-swap2-fixpoint-10.qdimacs	319.853	1.787	MEM	0.047	sat
small-swap2-fixpoint-2.qdimacs	7.007	0.31	MEM	0.016	unsat
small-swap2-fixpoint-3.qdimacs	38.127	0.45	MEM	0.016	sat
small-swap2-fixpoint-4.qdimacs	40.767	0.543	MEM	0.016	sat
small-swap2-fixpoint-5.qdimacs	101.52	0.746	MEM	0.031	sat
small-swap2-fixpoint-6.qdimacs	81.994	0.836	MEM	0.031	sat
small-swap2-fixpoint-7.qdimacs	152.68	1.11	MEM	0.046	sat
small-swap2-fixpoint-8.qdimacs	188.513	1.267	MEM	0.046	sat
small-swap2-fixpoint-9.qdimacs	318.016	1.576	MEM	0.031	sat
small-synabs-fixpoint-1.qdimacs	2.2	0.197	0.523	0.016	unsat
small-synabs-fixpoint-10.qdimacs	7.203	329.67	MEM	0.093	unsat
small-synabs-fixpoint-2.qdimacs	1.843	0.563	MEM	0.016	unsat
small-synabs-fixpoint-3.qdimacs	2.273	1.806	MEM	0.016	unsat
small-synabs-fixpoint-4.qdimacs	2.83	2.117	MEM	0.031	unsat
small-synabs-fixpoint-5.qdimacs	3.693	4.03	MEM	0.046	unsat
small-synabs-fixpoint-6.qdimacs	3.887	17.686	MEM	0.047	unsat
small-synabs-fixpoint-7.qdimacs	5.437	26.947	MEM	0.062	unsat
small-synabs-fixpoint-8.qdimacs	7.447	61.907	MEM	0.062	unsat
small-synabs-fixpoint-9.qdimacs	6.907	76.893	MEM	0.078	unsat
usb-phy-fixpoint-1.qdimacs	229.333	2.163	MEM	0.187	unsat
usb-phy-fixpoint-2.qdimacs	TIME	50.123	MEM	1.388	unsat
usb-phy-fixpoint-3.qdimacs	TIME	14.86	MEM	2.496	unsat
usb-phy-fixpoint-4.qdimacs	TIME	TIME	MEM	5.491	unsat
usb-phy-fixpoint-5.qdimacs	TIME	TIME	MEM	7.753	unsat

TABLE I
EXPERIMENTS: HARDWARE FIXPOINT CHECKS

	sKizzo	QuBE	Quantor	Z3	Result
1394diag_ioctl.c.qdimacs	TIME	TIME	MEM	TIME	?
1394diag_isochapi.c.qdimacs	MEM	TIME	MEM	40.591	sat
audio_ac97_common.cpp.qdimacs	MEM	TIME	MEM	0.468	sat
audio_ac97_rtstream.cpp.qdimacs	MEM	TIME	MEM	0.202	sat
audio_ac97_wavepcistream.cpp.qdimacs	TIME	TIME	MEM	416.757	unsat
audio_ac97_wavepcistream2.cpp.qdimacs	MEM	TIME	MEM	0.483	unsat
audio_ac97_wavepcistream3.cpp.qdimacs	MEM	TIME	MEM	0.219	unsat
audio_ddksynth_csynth.cpp.qdimacs	MEM	TIME	MEM	0.358	unsat
audio_ddksynth_csynth2.cpp.qdimacs	MEM	TIME	MEM	0.094	sat
audio_ddksynth_voice.cpp.qdimacs	TIME	TIME	MEM	28.08	unsat
audio_dmusuart_mpu.cpp.qdimacs	TIME	TIME	MEM	34.179	sat
audio_fmsynth_miniport.cpp.qdimacs	MEM	TIME	MEM	0.156	sat
audio_fmsynth_miniport2.cpp.qdimacs	MEM	626.356	MEM	0.109	sat
audio_gfxswap.xp_filter.cpp.qdimacs	MEM	TIME	MEM	0.592	unsat
audio_sysfx_swap.cpp.qdimacs	MEM	TIME	MEM	TIME	?
AVStream_hwsim.cpp.qdimacs	TIME	TIME	MEM	TIME	?
AVStream_image.cpp.qdimacs	MEM	TIME	MEM	22.401	sat
filesys_cdfs_allocsup.c.qdimacs	MEM	TIME	TIME	TIME	?
filesys_cdfs_cddata.c.qdimacs	MEM	TIME	MEM	TIME	?
filesys_cdfs_namesup.c.qdimacs	MEM	TIME	MEM	TIME	?
filesys_cdfs_namesup2.c.qdimacs	MEM	TIME	MEM	0.14	sat
filesys_fastfat_allocsup.c.qdimacs	MEM	TIME	MEM	0.187	sat
filesys_fastfat_cachesup.c.qdimacs	MEM	TIME	MEM	0.202	sat
filesys_fastfat_easup.c.qdimacs	MEM	TIME	MEM	6.676	sat
filesys_fastfat_write.c.qdimacs	MEM	TIME	MEM	TIME	?
filesys_filter_namelookup.c.qdimacs	MEM	TIME	MEM	TIME	?
filesys_smbmrx_cvsndrcv.c.qdimacs	523.737	TIME	MEM	0.156	unsat
filesys_smbmrx_midatlas.c.qdimacs	40.877	TIME	MEM	0.047	unsat
filesys_smbmrx_smbxchng.c.qdimacs	MEM	TIME	MEM	55.816	unsat
general_pcidrv_sys_hw_eeprom.c.qdimacs	MEM	TIME	MEM	0.843	unsat
general_pcidrv_sys_hw_eeprom2.c.qdimacs	MEM	TIME	MEM	0.499	sat
general_toaster_exe_notify_notify.c.qdimacs	MEM	TIME	MEM	TIME	?
hid_firefly_app_firefly.cpp.qdimacs	TIME	TIME	MEM	TIME	?
hid_hclient_ecdisp.c.qdimacs	MEM	TIME	MEM	40.108	sat
input_mouser_cseries.c.qdimacs	MEM	TIME	MEM	0.421	sat
input_mouser_detect.c.qdimacs	MEM	1796.263	MEM	0.031	sat
input_pnpi8042_moudep.c.qdimacs	MEM	TIME	MEM	51.377	sat
ir_smscir_io.c.qdimacs	MEM	TIME	MEM	TIME	?
kernel_agplib_init.c.qdimacs	MEM	TIME	MEM	0.109	sat
kernel_agplib_intrface.c.qdimacs	MEM	TIME	MEM	0.187	sat
kernel_uagp35_gart.c.qdimacs	MEM	TIME	MEM	40.107	sat
kmdf_AMCC5933_sys_S5933DK1.c.qdimacs	MEM	TIME	MEM	0.141	sat
kmdf_osrusbfx2_exe_dump.c.qdimacs	MEM	TIME	MEM	47.411	unsat
kmdf_osrusbfx2_exe_testapp.c.qdimacs	MEM	TIME	MEM	TIME	?
kmdf_pcidrv_sys_hw_nic_init.c.qdimacs	MEM	TIME	MEM	38.016	sat
kmdf_pcidrv_sys_hw_physet.c.qdimacs	MEM	TIME	MEM	0.031	sat
kmdf_usbsamp_sys_queue.c.qdimacs	MEM	TIME	MEM	TIME	?
mmedia_gsm610_gsm610.c.qdimacs	MEM	TIME	MEM	0.187	sat
mmedia_gsm610_gsm6102.c.qdimacs	284.807	TIME	MEM	0.109	unsat
mmedia_gsm610_gsm6103.c.qdimacs	371.61	TIME	MEM	0.577	unsat
mmedia_imaadpcm_imaadpcm.c.qdimacs	MEM	TIME	MEM	TIME	?
network_irda_miniport_nscirda_comm.c.qdimacs	MEM	TIME	MEM	408.901	unsat
network_irda_miniport_nscirda_settings.c.qdimacs	MEM	TIME	MEM	515.143	unsat
network_ndis_coisdn_TpiParam.c.qdimacs	MEM	TIME	MEM	8.158	sat
network_ndis_e100bex_5x_kd_mp_dbg.c.qdimacs	MEM	TIME	MEM	TIME	?
network_ndis_rtlnwifi_extsta_st_aplst.c.qdimacs	MEM	TIME	MEM	42.028	sat
network_ndis_rtlnwifi_extsta_st_misc.c.qdimacs	TIME	TIME	MEM	TIME	?
network_ndis_rtlnwifi_hw_hw_ccmp.c.qdimacs	MEM	TIME	MEM	1.279	sat
network_trans_sys_notify.c.qdimacs	209.523	TIME	MEM	6.224	unsat

TABLE II
EXPERIMENTS: RANKING FUNCTION SYNTHESIS.