Journal on Satisfiability, Boolean Modeling and Computation 5 (2008) 133-191

QBF-Based Formal Verification: Experience and Perspectives

Marco Benedetti[*] marco.benedetti@univ-orleans.fr
Laboratoire d'Informatique Fondamentale d'Orléans, University of Orléans, France

Hratch Mangassarian hratch@eecg.toronto.edu
Electrical and Computer Engineering Department, University of Toronto, Canada

Abstract

The language of Quantified Boolean Formulas (QBF) has a lot of potential applications to Formal Verification (FV) tasks, as it captures many of these tasks in a natural and compact way. Practical experience has been disappointing though. When compared with contending approaches such as SAT, QBF-based FV has invariably yielded unfavorable experimental results.

This paper makes two contributions. We first provide an account of the status quo in QBF-based FV. We examine commonly adopted formalizations and the relative strengths of different decision procedures. In the second part of this paper, we investigate for the first time the relevance of some advanced QBF techniques to FV tasks. In particular, we describe the use and the benefits of restricted quantifiers, QBF certificates, alternative encodings for classical model checking problems, and encodings with free variables. These promising research perspectives seem to reverse the negative standing of QBF applied to FV, as confirmed by the experimental evidence we discuss. Experiments are conducted by extending the publicly available solver sKizzo in several ways, and they include the first case studies where QBF compares favorably to SAT, its traditional competitor. QBF turns out to be an order of magnitude faster than SAT in some tasks (e.g., automated design debugging of large circuits). Moreover, as the size of the problems grows, the SAT encodings result in excessive memory requirements leading to out-of-memory conditions, while the more compact QBF encodings continue to be manageable and solvable.

KEYWORDS: *quantified Boolean formulas, formal verification, empirical evaluation*

Submitted February 2007; revised April 2008; published June 2008

1. Introduction

Formal verification (FV) is concerned with proving or disproving the correctness of a system with respect to a certain property using formal methods. One of the most promising formalisms for FV applications is the language of Quantified Boolean Formulas (QBF). QBF shows great promise for two reasons. On the one hand, there have been major improvements in recent years for the simpler but closely related NP-complete problem of checking the satisfiability of propositional statements (PROP) coming from the encoding of FV tasks.

[*] This author is supported by the French research agency Le STUDIUM® (lestudium.cnrs-orleans.fr).

Satisfiability (SAT) solvers have been successfully used to address a large class of industrial-scale problems [33] in the area of computer-aided design of integrated circuits [48, 50] and Model Checking for dynamic systems [21], to name a few.

On the other hand, the more expressive language of QBF, which adds the valuable possibility to quantify—universally or existentially—over the truth value of each variable, captures the wider class of PSPACE-complete problems. This enables us to produce expressive and compact formulations of many formal verification tasks that would require a significantly larger description in PROP.

But, do QBF solvers add substantial value to the reasoning capabilities of SAT solvers as far as FV tasks are concerned? More broadly speaking, are they ready to become the reference technology in any class of FV tasks? Many results suggest that this is not yet the case [75, 59, 46, 45].

So far, solutions to FV problems based on propositional logic have seen their best embodiment in procedures relying on SAT solvers. Researchers have been confronted by heavy time/memory tradeoffs in their attempts to shift to more powerful formalisms, such as the QBF logic we consider here. For example, despite the ability of QBF to capture FV problems in "compressed" forms, these shorter formalizations have turned out to be more time-intensive to deal with than SAT-based ones.

It has not been fully understood if such a time/memory trade-off is inescapable, as some believe, or whether it can be bypassed in practical cases. It has also been unclear whether the problem can be resolved by improving existing solvers or by alternative decision procedures, or whether the issue is more about the way QBF encodings of FV problems are formulated than about the solving strategy.

This paper contributes to the above open research questions in two ways: (1) We provide the first thorough FV-oriented survey of the current state-of-the-art in QBF solvers, benchmarks, and encodings, and (2) we bring in concepts and findings that recently emerged in the QBF and related research communities, and we start to investigate their potential in FV. The paper is accordingly divided into two parts:

The Experience part (Section 2) looks at the current situation. After a basic introduction to QBF (Section 2.1), we report on the *status quo* of QBF-based formal verification in terms of commonly adopted formalizations (Section 2.2), established benchmarks in the public domain (Section 2.3), existing QBF decision procedures (Section 2.4) and their relative strengths compared to each other (Section 2.5) and compared to alternative SAT-based approaches (Section 2.6). Some new insights on how challenging instances are solved by state-of-the-art provers are given in Section 2.5.

The Perspectives part (Section 3) looks at the future. We introduce promising research directions which might reverse the current reputation of QBF as a technology unsuited to real-world applications. Recent results from related research communities are reinterpreted for QBF and from the perspective of FV applications, namely the *restricted quantification* technique (Section 3.1) recently introduced [19] to model and solve quantified constraint satisfaction problems (QCSPs), and the *alternative encodings* for bounded sequential modeling (Section 3.2) proposed by the design automation community [55, 56]. The relevance to FV of new QBF-specific techniques, such as *validity certification* [13] (Section 3.3) and encodings with *free variables* (Section 3.4), are investigated for the first time. We use the

QBF solver sKizzo [15] as a testbed to explore all these new perspectives. The extensions we implement are made publicly available at [16]. We show new experimental results on the certification of FV instances, on the relative performance of QBF solvers, and we report, for the first time, results favorable to QBF compared to SAT.

We conclude the paper in Section 4 by summarizing our findings and commenting on the central message of the paper, which is: Despite the invariably negative results of QBF applied to FV, many promising research directions exist which are likely to shake QBF out of the current impasse.

This paper is an updated and largely extended version of [18]. The material from [18], presented here in sections 2.3, 2.4, 2.5, 2.6, and in the first part of Section 3.3.2 has been updated by improving the presentation and by including experimental results for solvers which didn't exist when the original paper was prepared. The remaining material, roughly 70% of this paper, is original.

2. QBF in FV: The State of the Art

In this section we provide examples of what QBF can do in principle and how ineffective it has turned out to be in practice at formal verification tasks. We start by presenting basic definitions (Section 2.1), classical QBF formalizations of FV problems (Section 2.2), and some publicly-available QBF instances which have been obtained using these formalizations (Section 2.3). Then, we review different strategies and solvers developed to decide QBF (Section 2.4), and present experimental comparisons showing how these solvers react to the above mentioned instances (Section 2.5). Finally, we survey recent contributions comparing QBF-based FV with contending approaches, namely SAT-based ones (Section 2.6).

2.1 Quantified Boolean Formulas

We give an informal presentation and some basic notions about QBF, and refer the reader to specialized texts (e.g., [26]) for a thorough introduction to their semantics and properties. The quantified Boolean formulas we are interested in are logic statements like this:

$$\forall a \exists b \forall c \exists d.\ (\neg a \vee c \vee d) \wedge (\neg b \vee \neg d) \wedge (a \vee b \vee \neg d) \wedge (\neg a \vee b) \tag{1}$$

In this example formula, the Boolean (or propositional) variables a, b, c, d can be assigned to one of the truth values 1 (for "true") or 0 (for "false"). The *prefix* "$\forall a \exists b \forall c \exists d$", read left-to-right, instructs us to consider the whole formula as true if and only if for both truth values of a it is possible to assign a truth value to b such that for both truth values of c it is possible to assign a truth value to d such that the *matrix* "$(\neg a \vee c \vee d) \wedge (\neg b \vee \neg d) \wedge (a \vee b \vee \neg d) \wedge (\neg a \vee b)$" is *satisfied* in the propositional sense, i.e., at least one literal in each of the four clauses is true.

Each maximal subsequence of variables with the same type of quantifier in the prefix is called a *scope*. The number of *alternations* in a formula is the number of scopes in its prefix minus one. In (1) there are four scopes, each one containing one variable only, and three alternations. The *outermost* (*innermost*) scope is the first (last) one along the prefix in the left-to-right direction. The variable x *dominates* the variable y if x appears to the left of y in the prefix.

The QBF (1) is in *prenex normal* form, as all the quantifiers are grouped in front of a quantifier-free formula in conjunctive normal form. This is the standard input format for the majority of solvers. Most of the QBFs we write in this paper are not in prenex normal form. Their matrix may be an arbitrary propositional formula rather than a set of clauses; or, some quantifiers may appear inside the formula rather than in front of it. Such arbitrary formulas can, without loss of generality, be rewritten in prenex normal form by polynomial-time translations [65, 9, 37], and then be fed to standard QBF solvers. Tools devoted to automate such translations exist (e.g., `qst` [85]).

All the variables in the matrix of (1) are quantified somewhere in the prefix. This means that the formula is *closed*. Current QBF solvers take as input closed formulas only (non-quantified variables, if present in the input formula, are arbitrarily displaced in the innermost or outermost existential scope), and yield back a yes/no answer on whether the formula evaluates to true, plus some additional information on how to assign variables in the outermost scope. Formulas containing non-quantified (or *free*) variables are discussed in Section 3.4.1. Some of the decision procedures adopted by present solvers are discussed in Section 2.4.

2.2 QBF Encodings of some Classical FV Problems

Many FV tasks exist for which QBF formulations are natural (although being natural does not imply being easy to solve). As an example, we review the QBF encodings of *model checking* (MC) and *diameter computation* problems for finite-state systems. The list of encodings we present is just meant as a frame for the results in the subsequent sections, and is by no means comprehensive.

2.2.1 Model Checking for Reachability Properties

Suppose the state of a system is represented by a state vector s made up of n bits, while its dynamics are captured by a Boolean Kripke structure we represent as a couple (I, T), where $I(s)$ is a predicate that recognizes initial states, and the *transition relation* $T(s, s')$ tells valid transitions (from the current state s into a next state s') from invalid ones. A sequence of states $s_0, s_1, \ldots, s_n$ such that the transitions from s_i to s_{i+1} are valid for $i = 0, \ldots, n-1$ is a *valid path* from s_0 to s_n. If a valid path from s to s' exists, then s' is *reachable* from s.

One classical formal verification problem for systems represented as Kripke structures is the model checking problem for *safety* properties, also called *reachability* problem [57, 32]. Given a Kripke model for a system and a set of *bad states* for that system, the reachability problem asks: *Is it true that no bad state is reachable from initial states?*

2.2.2 SAT-Based Bounded Model Checking

No a priori bound is put on the length of the paths that might be involved in (dis)proving reachability for bad states. For this reason, the problem is sometimes called *unbounded* model checking. The *bounded* model checking (BMC) variant asks whether a bad state can be reached *in k (or less) steps*. By bounding the horizon of reasoning, we lose the ability to prove safety in the absolute sense, but we gain the possibility to express BMC as a SAT problem, as shown in [21]. In particular, the existence of a valid path of length k, rooted at some initial state s_0 and terminating in some bad state s_k, is associated with the satisfiability of the formula

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge B(s_k) \tag{2}$$

where $B(s)$ captures the bad states. Note that the transition relation is replicated once for each transition in the path (we say that T has been *unrolled* k times). The formula (2) is to be tackled by SAT solvers. If it is satisfiable, its models encode *counterexamples*, i.e., k-step long sequences of valid transitions leading the system from an initial state into a bad state[1.].

BMC is widely employed in practice to discover undesired behaviors (bugs) of systems [33]. However, when the systems represented by the transition relation are complex (tens of thousands of clauses) and the depth of analysis is high (thousands of steps), the formulation (2) inevitably strains memory resources, up to the point where SAT-based model checking becomes impossible.

2.2.3 QBF-Based Bounded Model Checking

QBF-based approaches to BMC have been among the first to be explored by the formal verification community, because valid paths of length k can be characterized in QBF without unrolling the transition relation multiple times [67, 59, 20, 35, 46, 45]. To this end, modeling techniques known since a long time [74, 80, 64] can be adopted. For example, we leverage universal quantifiers to express validity as follows: A sequence of states describes a valid path if and only if "for any two states x and x', if x and x' are adjacent along the sequence, then they are consistent with the transition relation". For a sequence of states $s_0, s_1, \ldots, s_n$ this is formally written in QBF[2.] as:

$$\forall x, x' \; [\vee_{i=0}^{k-1}(x = s_i) \wedge (x' = s_{i+1})] \rightarrow T(x, x') \tag{3}$$

This formula contains only one copy of the transition relation (the biggest component in most specifications) no matter how many steps the path is comprised of. Once valid paths are described in QBF, the BMC problem itself is readily captured (again, using a single copy of T) as:

$$\exists s_0, \ldots, s_k \; I(s_0) \wedge B(s_k) \wedge \forall x, x' \; [\vee_{i=0}^{k-1}(x = s_i) \wedge (x' = s_{i+1})] \rightarrow T(x, x') \tag{4}$$

The fact that (4) contains only one copy of T while (2) contains as many copies as there are transitions in the path, implies that for any non-trivial system the size of (2) increases with k much faster than the size of (4). A smaller encoding might lead us to expect run-time advantages as a corollary. As we shall discuss in the following sections, such a run-time advantage has not materialized so far.

A number of universal variables proportional to the number of state bits in the system are used in (3−4). A reformulation of (3) exists which uses k auxiliary Boolean variables

1. The path we consider in (2) is exactly k-step long. The "at most k steps" variant is obtained by replacing $B(s_k)$ with $\vee_{i=0}^{k} B(s_i)$ or by augmenting the transition relation with self-loops at each state: $T'(s, s') \equiv (s = s') \vee T(s, s')$.
2. The notation $x = y$, where $x = \langle x_1, \ldots, x_n \rangle$ and $y = \langle y_1, \ldots, y_n \rangle$ are n-bit vectors, is a shorthand for $\wedge_{i=1}^{n} x_i \leftrightarrow y_i$.

$\alpha_1, \ldots, \alpha_k$ independently of the size of the description for system states. To obtain such a reformulation we introduce a formula $one(\alpha_1, \ldots, \alpha_k)$ which recognizes when exactly one of its Boolean arguments is true:

$$one(\alpha_1, \ldots, \alpha_k) \equiv \bigvee_{i=1,\ldots,k} \alpha_i \wedge \bigwedge_{\substack{i,j=1,\ldots,k \\ i \neq j}} (\neg\alpha_i \vee \neg\alpha_j) \tag{5}$$

The simple expression (5) requires quadratic space (in k) to capture the meaning of *one*, yet linear formulations exist, as discussed in [35, 45]. Using *one*, valid paths are captured as:

$$\forall\alpha_1, \ldots, \alpha_k\ \exists x, x'\ T(x, x') \wedge [one(\alpha_1, \ldots, \alpha_k) \rightarrow \wedge_{i=1}^{k}(\alpha_i \rightarrow (x = s_{i-1}\ \wedge\ x' = s_i))] \tag{6}$$

Auxiliary universal variables are utilized in (6) as selectors of transitions along the path: The variable α_i selects the i^{th} transition. All the combinations of transitions are considered due to the universal quantification on the auxiliary variables. When multiple transitions are simultaneously selected, *one* falsifies the premise of the implication in (6). So, only one individual transition at a time is considered, and one copy of the transition relation suffices to express its validity.

A different technique is used in the *iterative squaring* encoding, in which the reachability $T^{2k}(s, s')$ of s' from s in $2k$ steps is defined as the existence of an intermediate state m which is reachable from s in k steps, and from which s' can be reached, again in k steps:

$$T^{2k}(s, s') \equiv \exists m\ T^k(s, m) \wedge T^k(m, s') \tag{7}$$

where $T^1(x, x') \equiv T(x, x')$. At each application of this iterative squaring rule we double the number of transitions taken into account, but also the number of copies of the transition relation that appear in the final formula. In this sense, formulation (7) is essentially equivalent to (2). Fortunately, the intervention of universal quantifiers can stop the proliferation of the copies of T, without altering the "iterative squaring" nature of the rule. In particular, universal quantifiers are leveraged to express the validity of the two half-paths in T^{2k} (the path from s to m and the path from m to s') by the same copy of T^k. For this reason the technique is called *non-copying* iterative squaring [74, 80]:

$$T^{2k}(s, s') \equiv \exists m\ \forall x, x'\ [(x = s \wedge x' = m) \vee (x = m \wedge x' = s')] \rightarrow T^k(x, x') \tag{8}$$

This formula shares a key feature with (4) and (6): Only one copy of the transition relation is required for any length of the path because T^{2k} mentions T^k only once, so by induction T^1 is present in the complete formula only once. This improvement comes at the cost of introducing two quantifier alternations per application, compared to the fixed $\forall\exists$ alternation in (6).

The number of universal variables introduced in (8) is proportional to the number of state bits in the system, as in (3). Just like (6) improves over (3) by disengaging the number of universals from the size of the states, (8) can be rephrased to use far less universal variables. The quantifiers "$\forall x, x'$" span over all the couples of state configurations. There are in fact only two relevant cases: $(x = s \wedge x' = m)$ and $(x = m \wedge x' = s')$. These two alternatives can be associated with the two truth values of a single universal variable, which is used to switch between two half-paths:

$$T^{2k}(s,s') \equiv \exists m\ \forall \alpha\ \exists x, x'\ T^k(x,x') \wedge [\alpha \rightarrow (x = s \wedge x' = m)] \wedge [\neg\alpha \rightarrow (x = m \wedge x' = s')] \quad (9)$$

The latter formulation introduces only one new universal variable (and quantifier alternation) per application, so the overall number of universals is logarithmic in the length of the path (i.e., it is exponentially less than in (6) for the same k). And, only one copy of the transition relation is used.

2.2.4 QBF-Based Unbounded Model Checking

The rapid growth of the reasoning horizon in the iterative squaring method could turn this technique into a complete MC method, once some upper bound on the length of potential counterexamples is exceeded. The diameter (or eccentricity) of the system—defined as the length of the longest among the shortest paths between any two states—is one such upper bound: If any two states can be connected, then they can be connected by a path whose length does not exceed the value of the diameter. This property holds in particular for an initial state and a bad state, so if a counterexample exists, it will be encountered before the model checking bound surpasses the diameter.

Tighter upper bounds can be defined. For example, in the *initialized* diameter only paths rooted at some initial state are considered (i.e., the longest among the shortest paths between an initial state and an arbitrary state is considered). It follows that the initialized diameter is not greater than the diameter (it may be much smaller in practice), but it is still an upper bound on the length of counterexamples, because all the potential counterexamples are rooted at some initial state.

QBF formulations can be used to estimate the (initialized) diameter of a system [59, 81, 45]. Let us consider, for example, the QBF formula:

$$\exists s_0, \ldots, s_k\ [\ P^k(s_0, \ldots, s_k) \wedge \forall x_0, \ldots, x_{k-1}(P^{k-1}(x_0, \ldots, x_{k-1}) \rightarrow \wedge_{i=0}^{k-1} \neg(s_k = x_i))\ ] \quad (10)$$

where $P^k(x_0, \ldots, x_k) \equiv I(x_0) \wedge T(x_0, x_1) \wedge \ldots \wedge T(x_{k-1}, x_k)$ characterizes valid initialized paths of length k (we could have used of course a compact QBF-based formulation like (4) to capture the meaning of P^k). The formula (10) captures the existence of an initialized path of length k whose last state is not traversed or reached by any initialized path of length $k-1$. The existence of such path in turn implies that the initialized diameter of the system is at least k.

Another technique to prove that bad states can never be reached, which does not require a check against pre-computed upper bounds (conversely, it can be used to *provide* upper bounds, as we shall see), is called *k-induction*, and works as follows [76]: If we can prove that the system never traverses or reaches bad states in any path of length k provided that the path is rooted at a good state (we call this "k-induction check"), then we know by induction that the system will always stay in good states, assuming the initial states are all good. Note that the opposite is not true, i.e., the k-induction check may fail (for any k smaller than the system diameter) despite the safety of the system. The reason is that the paths considered in the k-induction check are arbitrary paths: They are not necessarily rooted at initial states, and they are not even guaranteed to be rooted at states that can be reached from the initial conditions. A path leading to a bad state poses no danger to the system if it is rooted at a non-reachable state. Yet, it falsifies the k-induction condition.

The k-induction check can be formulated as a SAT instance by reversing the perspective: We show that no k-step long path exists which traverses $k-1$ *different* good states and ends up in a bad state. The key point in this technique is to characterize *loop-free* (or *simple*) *paths*, i.e., paths which never traverse the same state more than once. Loop-free paths can be captured by the straightforward formalization $\wedge_{0 \le i < j \le k} \neg(s_i = s_j)$, which demands quadratic space in the length of the path. Once again, the QBF formulation is more compact [45], as it only demands linear space:

$$\forall \alpha_0, \ldots, \alpha_k \; \exists x \; [one(\alpha_0, \ldots, \alpha_k) \rightarrow \wedge_{i=0}^{k} (\alpha_i \leftrightarrow (x = s_i))] \tag{11}$$

In this formula, we leverage the predicate *one* to say, literally: "for every state s_i in the path, there exists a state x which is equal to s_i and different from all the other s_j, $j \neq i$", and this implies that if a state is traversed at step i in the path, it is not traversed before or after step i, hence the path is loop-free. For (11) to work in linear space, a convenient formulation of *one* is required, as in [45].

Once the validity of paths and the absence of loops are expressed in QBF using (6) and (11), it makes sense to capture the entire k-induction check via QBF. The missing part is the description of a path $s_0, \ldots, s_k$ that traverses the good states $s_0, \ldots, s_{k-1}$ and terminates in the bad state s_k. Using the same technique as in (6) and (11), this requirement is expressed in QBF as:

$$B(s_k) \wedge \forall \alpha_1, \ldots, \alpha_k \; \exists x \; \neg B(x) \wedge [one(\alpha_1, \ldots, \alpha_k) \rightarrow \wedge_{i=1}^{k} (\alpha_i \rightarrow (x = s_{i-1}))] \tag{12}$$

Finally, by requiring that conditions (6), (11), and (12) hold at once, we obtain a pure QBF check for k-induction. By conjoining those three expressions and simplifying the result we obtain:

$$\begin{aligned} \exists s_0, \ldots, s_k \{ \; & B(s_k) \;\wedge\; \forall \alpha_1, \ldots, \alpha_k \; \exists x, x' \; T(x, x') \;\wedge\; \neg B(x) \;\wedge \\ & \wedge \big[one(\alpha_1, \ldots, \alpha_k) \rightarrow \wedge_{i=1}^{k} (\alpha_i \leftrightarrow (x = s_{i-1}) \;\wedge\; \alpha_i \rightarrow (x' = s_i)) \big] \} \end{aligned} \tag{13}$$

which compactly characterizes the existence of valid loop-free paths that traverse k (different) good states and then reach a bad state. Notice that (6), (11), and (12) use auxiliary universal variables and the function *one* in essentially the same way, so this shared syntactic structure has been factorized and appears only once in (13). The formula has also been simplified by not predicating loop-freeness over the last state. Such state is indeed the only bad one, and this implies that it cannot coincide with any of the previous good states.

The non-existence of k-step loop-free paths terminating into bad states, i.e., the fact that (13) evaluates to false, implies by induction that all the paths longer than k cannot traverse bas states, and this holds in particular for paths rooted at initial states. So, counterexamples longer than k cannot exist if (13) is false. Thus, a complete MC procedure for safety properties fully based on QBF can be obtained by checking (4) for increasing bounds, until (13) becomes false.

2.3 Some Publicly-Available QBF Instances from FV Domains

In the last few years, many instances of FV problems encoded as QBF were contributed to the QBFLIB's archive [42] (which contains QBF families from many domains, not just

FV). Quite often, these instance are contributed exactly to show what QBF solvers are *not yet* able to solve. As a consequence, many families are challenging for state-of-the-art QBF reasoners. Here we consider some now classic FV benchmarks, most of which are still regarded as hard although they have been included for many years. To get the feeling of what such instances look like see Table 1.

Ayari's benchmarks [7] (72 instances, 5 families) are obtained from real-world verification problems on circuits (*adder*, *DFlipFlop*, and *VonN*) and protocol descriptions (*MutexP* and *SzymanskyP*). Most of these benchmarks are quite challenging for modern solvers. Some of them (e.g., the "adder" series) have never been completely solved [51, 61].

Biere's benchmarks [20] (64 instances, 4 families) represent model checking problems (stating invalid safety properties over n-bit counters) encoded via the iterative-squaring method. Such problems are easy for BDD-based symbolic MC. Conversely, they are rather difficult for SAT-based bounded MC techniques, as they capture the worst-case scenario in which the number of steps necessary to falsify the property equals the diameter of the system. The resulting instances are hard for current QBF solvers [51, 61]. They have been used in [20] to show that "*[QBF-based BMC] can barely keep up with SAT-based BMC*" (quoted from [45]).

Katz's benchmarks [46] (16 instances, 2 families) encode the (symbolic) reachability problem for some hardware circuits using both formulation (3) and formulation (8), and are considered difficult for present solvers [51, 61].

Lahiri and Seshia's family (3 instances) encodes convergence checking problems, generated from term-level model checking [24]. None of these instances has ever been solved [51, 61].

Ling's benchmarks [54] (2 families, 8 instances) encode FPGA (Field Programmable Gate Array) logic synthesis problems as QBF instances, where the aim of the encoding is to determine whether a specific logic function can be implemented in a given programmable circuit.

Mneimneh and Sakallah's benchmarks [59] (12 families, 90 instances) encode the problem of computing the diameter (or sequential depth) for twelve of the ISCAS89 circuits, using a variant of equation (10). For each circuit, a sequence of related QBF instances is generated, the n-th of which checks whether that circuit has a new state at depth n. According to [59], the number of alternations in the whole group of families is equal to 2. These benchmarks are considered to be very difficult [59, 51, 61] (only 20% of the instances have been solved).

Pan's family [63] (5 instances) encode the existence of a suited output configuration for any input configuration of a barrel-shifter with n control bits and 2^n input lines. Together with the *mutex* family from Ayari's benchmarks, this family has a single $\forall\exists$ alternation.

Scholl and Becker's benchmarks [75] (8 families, 64 instances) encode formal equivalence checking of partial implementations of VLSI circuits coming from real-world designs, in which faults have been randomly inserted. The technique used to encode these problems has been presented many years ago, but these benchmarks are still quite challenging [51, 61].

Table 1. Some families of QBF instances from FV domains. Each row represents a family. *Alt.* denotes the number of quantifier alternations in the prefix. Each $Min - Max$ entry gives the minimal/maximal number of *Variables*/*Clauses*/*Alternations* respectively. The last column gives the shape of the most complex prefix, where the most complex prefix is the one with the highest number of alternations, or—that being equal—the one containing more variables. For example, the value ∃[88]∀[10]∃[14], relative to *fpga_F*, means that the prefix with more alternations in this family has 3 scopes, which are, in the left-to-right direction: an existential scope with 88 variables, a universal scope with 10 variables, and an existential scope with 14 variables.

	Family	#	*Variables*	*Clauses*	*Alt.*	*Shape of the most complex prefix*
Ayari	*adder-s*	8	332 – 6282	113 – 9528	3	∀[800]∃[872]∀[872]∃[2224]
.	*adder-u*	8	334 – 6312	114 – 9543	2	∃[1672]∀[872]∃[2254]
.	*Adder2s*	8	517 – 22329	292 – 25303	5	∀[512]∃[256]∀[16]∃[888]∀[872]∃[18271]
.	*Adder2u*	8	517 – 22329	291 – 25288	6	∃[512]∀[256]∃[16]∀[16]∃[872]∀[872]∃[18271]
	FlipFlop	9	551 – 160999	203 – 212621	2	∃[33]∀[55]∃[159748]
	MutexP	8	559 – 31177	127 – 14995	1	∀[512]∃[11058]
	Szyman.	10	3257 – 212561	451 – 242839	2	∃[22]∀[4180]∃[195205]
	VonN	9	25694 – 1013039	35189 – 1482992	2	∃[41]∀[420]∃[1007972]
Biere	*cnt*	16	16 – 1666	36 – 4401	2 – 32	∃[99]∀∃[99]∀∃[99]∀∃[99]...∀∃[99]∀∃[66]
.	*cnt_r*	16	21 – 1716	45 – 4545	2 – 32	∃[102]∀∃[102]∀∃[102]...∀∃[102]∀∃[68]
.	*cnt_e*	16	21 – 1716	[46 – 4561	2 – 32	∃[102]∀∃[102]∀∃[102]...∀∃[102]∀∃[68]
.	*cnt_re*	16	26 – 1766	55 – 4705	2 – 32	∃[105]∀∃[105]∀∃[105]...∀∃[105]∀∃[70]
Katz	*quant*	8	349 – 8276	643 – 15473	2	∃[793]∀[320]∃[4538]
.	*quantsqr*	8	349 – 9875	643 – 17713	2 – 8	∃[34]∀[22]∃[98]∀[22]∃[98]...∀[22]∃[140]
Lah./Sesh.	*uclid*	3	1464 – 5764	3839 – 15954	3 – 5	∀[42]∃[6]∀[216]∃[18]∀[6]∃[5476]
Ling	*fpga_F*	5	43 – 112	168 – 1832	2	∃[88]∀[10]∃[14]
.	*fpga_S*	3	70 – 75	446 – 675	2	∃[56]∀[8]∃[11]
Mneimneh	*s27*	4	85 – 403	142 – 478	2	∃[68]∀[49]∃[104]
&	*s298*	8	4744 – 78848	6482 – 30946	2	∃[2450]∀[2306]∃[7020]
Sakallah	*s386*	8	2478 – 36462	4811 – 24215	2	∃[1914]∀[1743]∃[5412]
.	*s499*	8	1213 – 90633	2665 – 39232	2	∃[2929]∀[2732]∃[8838]
.	*s510*	8	45786 – 801216	29679 – 132717	2	∃[11379]∀[11144]∃[30500]
.	*s641*	8	2539 – 56647	4494 – 36984	2	∃[4319]∀[3891]∃[8631]
	s713	8	2663 – 50519	4872 – 36232	2	∃[4069]∀[3626]∃[8388]
	s820	8	4769 – 97235	9640 – 57270	2	∃[4037]∀[3739]∃[12662]
	s1196	6	3686 – 38966	7939 – 40079	2	∃[3728]∀[3162]∃[8952]
	s1269	8	3985 – 185257	8494 – 97674	2	∃[8707]∀[8055]∃[22344]
	s3271	8	10749 – 1050099	22447 – 384227	2	∃[35523]∀[33169]∃[89061]
	s3330	8	11892 – 394318	23365 – 230033	2	∃[22960]∀[20940]∃[53328]
Pan	*q-shifter*	5	19 – 520	128 – 131072	1	∀[264]∃[256]
Scholl	*C432*	8	568 – 623	1439 – 1557	2 – 16	∃∀∃∀∃[2]∀∃[10]∀[2]∃[4]∀[2]E...∀[3]∃[574]
&	*C499*	8	838 – 906	2393 – 2586	2 – 18	∃[2]∀∃[2]∀∃[4]∀∃[2]AE∀∃[8]...∀∃[859]
Becker	*C880*	8	975 – 1046	2484 – 2644	2 – 20	∃[3]∀∃[2]∀∃[9]∀[2]∃[2]∀∃[2]...∀[3]∃[983]
.	*C5315*	8	5326 – 5606	14228 – 15230	2 – 30	∃[2]∀∃[2]∀[2]∃[2]∀∃[6]...∀[3]∃[5274]
.	*C6288*	8	4638 – 4808	13583 – 14005	2 – 14	∃[16]∀[10]∃[14]∀[13]∃[2]...∀[13]∃[4653]
.	*comp*	8	277 – 311	746 – 844	2 – 6	∃[2]∀∃[8]∀[2]∃[2]∀∃[295]
	term1	8	1020 – 1117	3526 – 3879	2 – 22	∃[3]∀∃[3]∀∃[3]∀∃[3]∀∃[3]...∀∃[1051]
	z4ml	8	61 – 66	185 – 200	2 – 4	∃[2]∀∃[2]∀∃[59]

These families cover a wide variety of FV problems and encoding techniques. They constitute our first set of benchmarks, used in the experiments of Section 2.5, where the decision procedures introduced in Section 2.4 are compared. Further benchmarks will be introduced in Section 3.

2.4 Decision Procedures and State-of-the-Art Solvers for QBF

Several different paradigms have been proposed to evaluate quantified Boolean formulas. Most approaches leverage revised versions of techniques originally developed for SAT solvers. For example, *search-based* solvers extend the DPLL-technique [34] to deal with universal quantification [29, 67, 30]. Models are searched for in the most natural way: Following the left-to-right order of the variables in the prefix during a top-down, depth-first visit of the semantic evaluation tree of the formula. This is by far the most common approach to QBF evaluation. In the solver evaluation reported in [52], held in 2003, all the competitive QBF solvers (such as Qsat [67], Qsolve [38], Quaffle [84], QuBE [43], semprop [53]) were search-based. The research on search-based QBF solvers is still very active today. Proposals range from solvers capable of advanced form of on-the-fly inferences based, for example, on binary clauses (2clsQ [71]) to the employment of combined DNF/CNF representations [83, 69].

Alternative solving paradigms are emerging [51, 61]. *Resolution-based* solvers, for example, build upon generalizations of resolution to the quantified case (*q-resolution* [25, 26]), and reverse the order in which quantifiers are considered: Rather than searching for models following the left-to-right prefix direction, they attempt to *solve* the formula by applying a refutationally complete inference procedure which removes quantifiers right-to-left while preserving validity (this method is called quantifier elimination). For example, quantor [20] applies a variant of quantifier elimination which works as follows. At each step, either an existential variables (and its associated quantifier) are removed from the last (innermost) existential scope by a proper sequence of q-resolutions steps, or a universal variable is removed from the last but one scope by means of *expansion* [20].

Some solvers employ compact representations for clause sets or sets of assignments, based on BDDs or ZDDs, in the spirit of [31]. In the SAT case, these so-called *symbolic solvers* show a certain strength on specific classes of instances, but seem to be not competitive in general [62]. Things change in the QBF framework. Both the idea of compressed/symbolic representations, and the shift from searching to solving seem to be promising [40]. Compressed representations can be adopted to support both search (ZQSAT [40]) and resolution (QMRES [63], QBDD [63]).

SAT-based solvers partially expand the meaning of a QBF instance into a SAT instance, which is then addressed by state-of-the-art SAT solvers. This practice, already proposed by the seminal paper [29] in the form of *trivial truth/falsity* tests, has been further investigated as a means to selectively expand quantified subformulas (QuBOS [8]), to reason on the propositional skolemization of instances [11], or to relax the left-to-right constraint on variable ordering (SQBF [70]).

Finally, *skolemization-based* solvers replace the original problem with the validity problem on the skolemized instance: Existential variables are replaced by skolem function symbols whose definition domains are chosen so as to preserve satisfiability (sKizzo [11, 12]).

In the following sections, we exercise some publicly available implementations of the above mentioned approaches against instances from FV tasks. Namely, we consider:

QuBE-LRN [43], v. 1.3, a search-based solver featuring lazy data structures for unit clause and pure literal propagation, plus conflict and solution learning.

SEMPROP [53], v. 2004-01-06, a search-based solver which includes dependency-directed backtracking and mechanisms to cache lemmas/models.

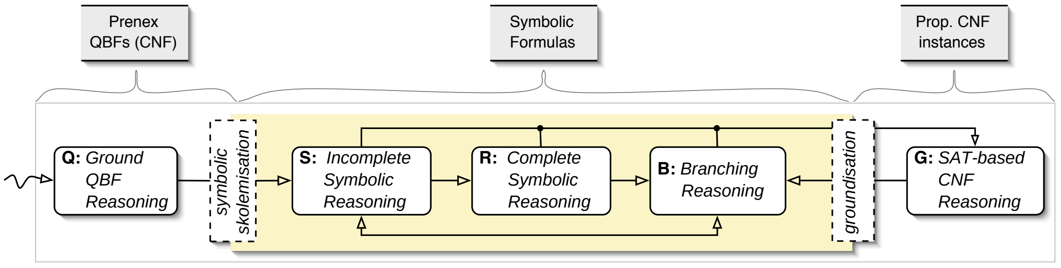


Figure 1. Block diagram of the *inference state machine* at the hearth of sKizzo.

yQuaffle [84], v. 2006-02-10, a search-based solver featuring multiple conflict-driven learning, inversion of quantifiers and solution-based backtracking.

SQBF [70], v. 2006-12-18, a search-based solver based on a bi-directional cooperation with a SAT solver (to exchange learnt clauses, etc.) with tight integration at the data-structure level.

2clsQ [71], v. 2006-12-18, a search-based solver employing a dynamic version of binary clause reasoning as a recursive form of forward inference.

Quantor [20], v. 2.11, a resolution-based solver employing q-resolution and expansion to eliminate quantifiers, subsumption control, plus a number of optimizations to improve efficiency.

sKizzo [15], v. 0.10, a hybrid solver based on symbolic forms of skolemization and clause representation, which performs resolution, search, and expansion to SAT.

Beyond the raw result of the experimental comparison, we provide in Section 2.5 a brief analysis on which method for inference turns out to be the best one for each set of benchmarks. This analysis is obtained by inspecting the *inference log*[3.] of sKizzo, and by letting it solve instances in different configurations. sKizzo is used as a framework to implement all the extensions we introduce in Section 3[4.]. For these reasons, we devote the next section to describe how this specific reasoner works. We limit ourselves to a brief exposition of the terms and techniques required to understand the analyses presented in the rest of the paper. For a comprehensive account of the algorithms and techniques used by the solver, we refer the reader to [11, 12, 13, 14, 15].

2.4.1 The QBF Solver sKizzo

sKizzo is a QBF solver based on a particular form of skolemization that works with a symbolic (BDD-based) representation for clauses, on top of which it applies top-down DPLL-like branching reasoning (in the spirit of [29, 30]), variants of quantified resolution (see [25]), and

3. The "inference log" is a collection of data the solver dumps to file on request. In the trace are recorded, chronologically, detailed information about inferences and other relevant operations performed to decide the instance. The evolution of time-dependent variables, such as the number of clauses in the current formula, or the amount of time spent in reordering decision diagrams, is also reported.

4. The use of sKizzo is not to be perceived as detrimental to the generality of the ideas presented in the paper. All the techniques we discuss in Section 3 can be handled by, e.g., search-based approaches.

techniques involving a (partial or total) expansion to SAT of the meaning of the formula (as described in [11]). These different approaches to the evaluation of QBFs are called *inference styles*, and are integrated and exercised within a coherent framework. For this reason, the solver is regarded as *hybrid*.

Inference styles are not applied to the original formula. Rather, they are exercised on its *symbolically skolemized* version, as defined in [12]. Symbolic skolemization is a translation that—applied to a QBF—yields a BDD-based representation of a set of clauses constraining acceptable interpretations for the skolem terms. This set of clauses is represented by means of a purpose-built data structure, described in [12], which tightly combines BDDs with classical list-based (or watched [60]) representations for clauses. The intuition is that BDDs take care of the universal side of the reasoning, while the existential side is dealt with using classical data-structures.

Besides allowing the integration of known inference styles for QBF, the symbolic skolemization framework enables a novel style of reasoning which operates right in the space of (interpretations for) skolem functions. *Symbolic rules* are defined in such space which are able to quickly simplify the formula (and, sometimes, to detect inconsistencies) by combining operations over BDDs with operations over lists of literals [12].

The solver attempts to sort out automatically the best inference styles, or to select and combine multiple approaches when appropriate, according to the *inference state machine* depicted in Figure 1. The solver encompasses three *representation spaces* and five different inference styles $S^{inf} = \{\mathsf{Q}, \mathsf{S}, \mathsf{R}, \mathsf{B}, \mathsf{G}\}$. Q is the initial style. Each transition $x \to y$ in Figure 1, $x, y \in S^{inf}$, is associated with a condition that triggers the shift from style x to style y. During such a transition, the target style y receives as input the current formula as provided by style x. Here we limit our description of the solver to some additional information for each inference state.

Q: Ground QBF Reasoning. In the Q-state sKizzo works in the original QBF space. The quantified form of three classical inference rules—*unit clause propagation*, *pure literal elimination*, and *forall-reduction*—is applied until fixpoint, i.e., until none of these rules can infer anything anymore. Then, a *quantifier tree* for the formula is reconstructed as described in [14]. Quantifier elimination is then applied to prune “cheap” leaves of the quantifier tree (a cheap leaf is one associated with a variable whose elimination cost does not exceed certain time/memory thresholds). The whole cycle is repeated until no cheap existential variable exists. Then, the solver moves to state S.

S: Incomplete Symbolic Reasoning. The instance undergoes *symbolic skolemization* [12] and is mapped onto an equivalent BDD-based *symbolic formula* (the CUDD package ver. 2.4.0 and DDDMP ver. 2.0.3 are used to manipulate and load/store BDDs). Then, the formula is attacked by means of a set of *symbolic inference rules*, designed to perform efficient symbolic deductions. These rules—applied according to a purpose-built scheduling algorithm described in [15]—are:

SUCP (Symbolic Unit Clause Propagation). Unit clauses are symbolically computed and assigned all at once in the space of skolem terms.

SPLE (Symbolic Pure Literal Elimination). A symbolic representation for the set of pure literals is computed, and the formula is accordingly simplified.

SSUB (Symbolic SUBsumption). All the symbolic clauses that are subsumed by other clauses are removed (*forward subsumption*). SSUB complements the *backward subsumption* mechanism applied on-the-fly at each clause insertion.

SBR (Symbolic Binary Reasoning). All the resolution chains of binary symbolic clauses in the formula are enumerated looking for contradictions, hence for failed symbolic literals to negate and assign.

SER (Symbolic Equivalency Reasoning). Strongly connected components in the *symbolic implication graph* [11] of the formula are identified. Each such component determines the application of a symbolic equivalence to simplify the formula.

The above rules are *refutationally incomplete.* They compute their deductive closure (without in general deciding the instance) and yield back a simplified equivalent instance. A refutationally complete rule (i.e., one that is expected to decide any instance, given enough time and memory) is also present in the pool of rules scheduled for application:

SDR (Symbolic Directional Resolution). This rule eliminates one symbolic variable per step by substituting the set of resolving clauses with the set of their symbolically computed resolvents.

SDR often generates too many clauses, and this may critically hamper the whole decision process. To keep this issue under control, SDR is applied in a controlled environment that triggers a rollback whenever the rule fails to shrink the formula (in terms of number of clauses) within pre-assigned time limits. The solver moves to state R when (1) all the incomplete rules have finished their computations and (2) the time-limited SDR rule is unable to shrink the formula.

R: Complete Symbolic Reasoning. This state is similar to S, with one major exception: No rollback is triggered if SDR enlarges the formula. So, incomplete rules are allowed to work on the enlarged SDR output. A rollback is triggered only if SDR generates so many clauses that main memory is exhausted. When this happens, the smallest formula obtained during the whole R computation is restored, then the solver switches to state B.

B: Branching Reasoning. In this state, a recursive search-based branching decision procedure extending the DPLL approach to QBF is applied. Both universal and existential splits are performed symbolically. The partial order induced by the internal structure of the quantifier tree constructed in Q is used to decide the order of splits (instead of the total left-to-right order of variables in the prefix). Either symbolic reasoning or ground reasoning are leveraged as look-ahead tools to decide base cases of the recursion. In particular, the incomplete rules of state S are applied recursively at each search node, and this may result in a decision without further splits. Similarly, the ground projection of the current formula may be small enough to be decided via expansion in the G style. A conflict-analysis process is triggered in the event of an inconsistent partial assignment to perform conflict-directed backjumping. The B, S, and G states share a common conflict-analysis engine. A symbolic learning mechanism extracts symbolic clauses from contradictions. Static and dynamic branching heuristics (MOMS, VSDIS) are used to guide the search.

G: SAT-based Ground Reasoning. In the G-state the solver constructs via an operation called *groundization* a SAT instance equivalent to the current subproblem and solves it using a SAT solver, which is one[5] of zChaff [60] (ver. 2004.5.13), siege (ver. 4), or minisat [36] (ver. 1.14). The data structures used in the solver support a very fast any-time estimation of the size of the equivalent SAT instance (which may suddenly change after inference steps). This feature allows the solver to enter the state G from every other state, as soon as SAT-based reasoning becomes a viable option.

sKizzo is capable of *satisfiability certification* [13], i.e., it reconstructs, for true formulas, an explicit representation of a strategy to satisfy the matrix. This feature has been introduced as a means to prove that the answers given by the solver are correct. However, the information conveyed by certificates may be exploited well beyond such basic task, and have interesting application to FV. For this reason, we review in the next section the basic properties of certificates. Their applications to FV will be discussed in Section 3.3.

2.4.2 Certificates of Satisfiability for QBFs

A *certificate of satisfiability* for a true QBF is any piece of information that provides self-supporting evidence of validity for that QBF. Several alternatives are possible as to what certificates contain and how they represent information [27, 13]. We adopt the version described in [13], where certificates are described as stand-alone, BDD-based, compact (but explicit) representations of the functional dependencies that have to exist between existential *dependent* variables and universal *independent* variables in order to satisfy the matrix for all the assignments to the universal variables.

The form of the dependencies in a certificate is restrained by the quantification prefix of the QBF the certificate refers to. In particular, the existential variable x may only depend on (a subset of) the universal variables that dominate x in the prefix. This restriction is a consequence of the intimate connection between the information carried by a certificate and the models of the *outer skolemized* version of the certified QBFs. In outer skolemization we remove each existential variable x by replacing it with a skolem term (or function) $s_x(u_1, \ldots, u_m)$ whose arguments $u_1, \ldots, u_m$ are the universal variables that dominate x. The resulting skolemized instance is equivalent to the originating QBF with respect to satisfiability, so that any representation for any model of the skolemized instance is a certificate of satisfiability for the originating QBF. Models of skolemized instances consist of the Boolean interpretations for their skolem terms: The dependencies encoded in a certificate are nothing but definable Boolean interpretation for all such skolem terms.

Example 1 *Let us consider the formula*

$$\begin{aligned}\forall a \forall b \exists c \forall d \exists e \exists f.(\neg b \vee e \vee f) \wedge (a \vee c \vee f) \wedge (a \vee d \vee e) \wedge (\neg a \vee \neg b \vee \neg d \vee e) \wedge (\neg a \vee b \vee \neg c) \wedge \\ \wedge (\neg a \vee \neg c \vee \neg f) \wedge (a \vee \neg d \vee \neg e) \wedge (\neg a \vee d \vee \neg e) \wedge (a \vee \neg e \vee \neg f)\end{aligned} \tag{14}$$

According to the prefix of (14), the skolem function associated with c is the term $s_c(a, b)$, while e and f are associated with the terms $s_e(a, b, d)$ and $s_f(a, b, d)$ respectively. So, we

5. sKizzo can be forced to use a specific SAT solver, although by default it decides autonomously which one to employ, depending on whether unsatisfiable cores are needed. The choice is between minisat (faster on average, but not able to extract unsatisfiable cores) and zChaff (slighlty less efficient, but capable of core extraction).

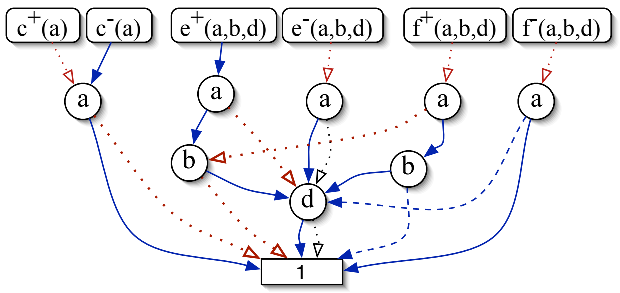


Figure 2. A certificate of satisfiability for the QBF (14). Details on how to read this diagram are in the text.

certify the validity of (14) by exhibiting valid *interpretations for s_c, s_e, and s_f, i.e., by exhibiting truth tables for s_c (as a function of a and b), s_e and s_f (as a function of a, b, d), such that any given assignment to the universal variables $\{a, b, d\}$, once extended with the assignment over $\{c, e, f\}$ computed according to the given truth tables, invariably satisfies the matrix of (14).*

A BDD-based[6.] compact representation for the truth tables of Example (1) is depicted in Figure 2 as a set of interconnected BDDs, called *forest*. Such a forest can be deciphered as follows.

- Each root node in the topmost line is associated with a total function (from assignments over universal/decision variables into $\{0, 1\}$) by the following mechanism: Any truth assignment A to the universal variables induces a unique path from any root node x to the sink node 1. Such a path is obtained by traversing *then-arcs* (solid lines) at decision levels associated with variables that are true in A, and *else-arcs* (dashed/dotted lines) at the other levels. While traversing such path, we count how many *complemented arcs* (dotted/red lines) are encountered. The function associated with the root node x evaluates to 1 under the assignment A if and only if an even number of complemented arcs is traversed;
- In the topmost line of the forest there are two root nodes for each skolem interpretation. For example, the interpretation of s_c is associated with the two nodes labeled c^+ and c^-. This is done to allow *don't-care conditions* to appear. We indeed consider three-valued Skolem interpretations that may evaluate to true (`T`), false (`F`), and don't-care (`DC`). A `DC` condition for a term s_x under the universal assignment A means that the existential variable x plays no role in satisfying the matrix, given A;
- To represent an interpretation (that admits don't-care conditions) for s_e, we combine the value of the two roots e^+ and e^- as follows:

$$s_e(\psi_1, \ldots, \psi_n) = \begin{cases} \mathtt{T} & \text{if } e^+(\psi_1, \ldots, \psi_n) = 1 \\ \mathtt{F} & \text{if } e^-(\psi_1, \ldots, \psi_n) = 1 \\ \mathtt{DC} & \text{if } e^+(\psi_1, \ldots, \psi_n) = 0 \text{ and } e^-(\psi_1, \ldots, \psi_n) = 0 \end{cases}$$

6. Many variants of BDDs are described in the literature. We adopt their *reduced ordered* version (ROBDDs) with *complemented arcs* [23, 78].

We note two properties of well-formed certificates for a QBF F: (a) they associate to e^+ and e^- two disjoint on-sets for every e, and (b) they contain no decision node on the universal variable u in the subgraphs rooted at e^+ and e^- if u does not dominate e in F.

A certificate can be verified against a QBF with no need to know how it was obtained. In this sense, certificates are solver-independent: The verification algorithm exploits the *evaluation* apparatus of the QBF logic, and does not perform deductions. Furthermore, the BDD nature of the certificate allows the entire check to be performed at the symbolic BDD level. For example, the algorithm presented in [13] efficiently checks a certificate against a QBF, clause by clause, using a few BDD-based operations.

sKizzo has been extended to allow for the extraction of certificates out of any true QBF it can manage to solve. The evaluation of the QBF and the certification of the answer are treated as two completely decoupled processes, interconnected through a textual *inference log*. The inference log is produced by the solver and read by an external *certifier* application (ozziKs) which is charged with interpreting the content of the log in order to construct and verify certificates.

Such an architecture is designed to work with any QBF solver, not just sKizzo. All the algorithms used to build and verify certificates are encapsulated in ozziKs, which can be used as a black-box, under the only requirement that the QBF solver dumps to the log, in some standard format, enough information on the inference steps it applies. In particular, ozziKs is at present able to interpret (and exploit to build certificates) the inference steps of decision procedures based on DPLL splits, unit clause propagation, compilation to SAT, quantified resolution, equivalence reasoning, and all the symbolic counterparts of these rules (see Section 2.4.1), under the assumption that these operations are documented in the log according to the format described at [13, 10].

2.5 Relative Strength of Different Decision Procedures on FV Instances

In this section we comment on the relative performance (solving time) of the state-of-the-art QBF solvers introduced in Section 2.4 on the FV instances described in Section 2.3. All the experiments, if not stated otherwise, have been run on a 2.6–GHz Xeon machine, with 2 GB of memory.

For each group of families, we compare the number of instances these solvers can decide as a function of the allotted time, up to 1000 seconds. This information is represented graphically as described in [41, 63]: The Y-value of a point in the plot gives the number of instances that can be solved within an amount of time represented by the X-value of the same point (given on a logarithmic scale). We also examine the way the hybrid solving engine inside sKizzo reacts to each single test set. This analysis gives us hints on how to push the performance further for the successful cases, or on how to improve on the families where results are relatively weak.

Ayari. Results for this group are reported in Figure 3. sKizzo and quantor are the best solvers, while search-based reasoners are consistently behind. This is a first evidence of a pattern confirmed by other experiments (presented next): For most FV families

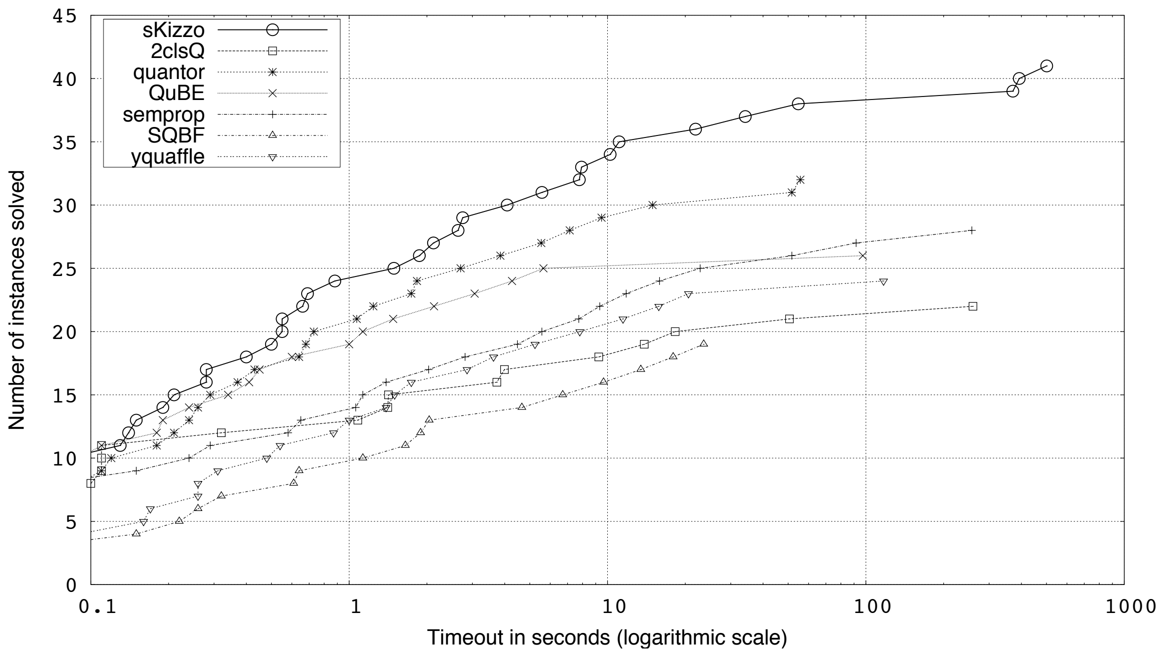


Figure 3. Performance of several QBF solvers over Ayari's benchmarks (72 instances).

search-based solvers seem not to be the best choice. The family of instances that contributes the most to sKizzo's performance is the *adder* one, especially the satisfiable series. This series is regarded as the hardest in the group. By analyzing the execution trace of the solver, we recognize two key elements that make a difference: (1) The quantifier tree reconstruction pre-processing [14] (in Q-state) is very effective in shrinking the number of universal quantifiers dominating each existential quantifier; (2) symbolic directional resolution [15], as made possible by symbolic skolemization (R-state in Figure 1), plays the main role in solving the formula. It is able to almost monotonically shrink the size of the instances until the empty formula is obtained. We succeeded for the first time in solving the whole *adder-s* series (and 6 out of 8 instances in the *Adder2s* series). The other families are easier to solve. In particular, the whole *VonN* series—despite hosting "monster" instances having more than one million variables—is completely solved in the Q-state (without even using q-resolution, i.e., the incomplete UCP/PLE rules suffice to evaluate the instance). The *MutexP* family is also solved in the Q-state, but ground q-resolution is the key operation this time. Not surprisingly, quantor—which uses essentially the same technique—is also very good at these instances. The more difficult cases come out to be the unsatisfiable adders. No solver decides more than a couple of them.

Biere. Figure 4 concerns the *counter* families. Again, search-based solvers lag behind alternative ones. quantor, the second-best solver, proceeds by quantifier elimination (see Section 2.4) until the entire theory is projected onto the outermost existential scope;

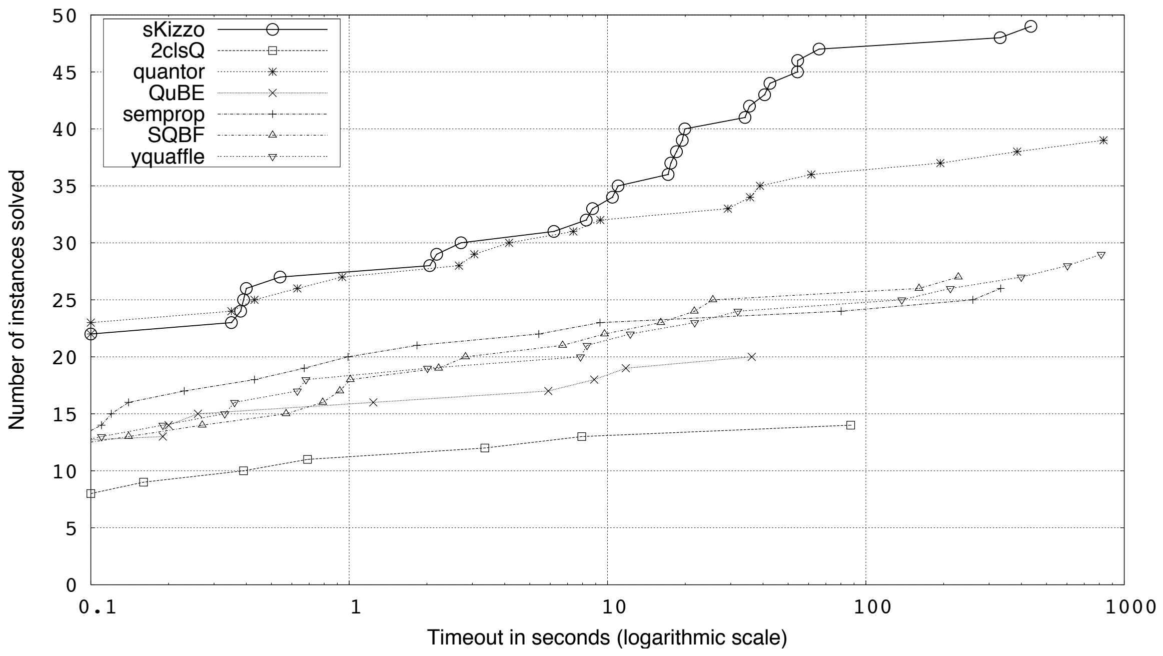


Figure 4. Performance of several QBF solvers over Biere's *counter* benchmarks (64 instances).

at that time quantor leverages a SAT solver as a back-end to decide the equivalent (and purely existential) instance (the instances are satisfiable, so no contradiction can be inferred along the way). sKizzo uses a completely different approach: It solves the counters in the G state. In particular, after symbolic normalization (state S) reaches its fixpoint, the instances are either completely solved (i.e., the entire interpretation of the skolem functions can be inferred without search; the whole *cnt* sub-family is solved this way), or their ground size has been reduced to the point that ground SAT-based reasoning is affordable. The experiments show that the bigger the instance, the more considerably G dominates S as to required time. In particular, S quickly delivers an expandable normalized form for all the instances in these families, yet solving the normalized form in G becomes more and more difficult as the size of the instance increases. So, what really matters is the efficiency of the underlying SAT solver. We have experimented with zChaff, siege, and minisat. The latter exhibits a sensible advantage over the former two (and is used in Figure 4). The use of other SAT solvers or some form of preprocessing of the SAT instance (such as the bounded variable elimination introduced by minisat, version 2.0) could be the key towards solving the whole group of instances. Currently, sKizzo solves 76% of the group within 1000 seconds, compared to 59% for the second-best solver.

Katz. These instances are relatively small but quite hard. No graph is given as no solver was able to decide anything. The reason for failure is always a timeout except for quantor and SQBF, which exhaust available memory. By inspecting sKizzo's trace, we observed that the problem is equivalence reasoning (SER in S) taking up all the time and resources. As opposed to what happens in SAT, a simplification by equivalence

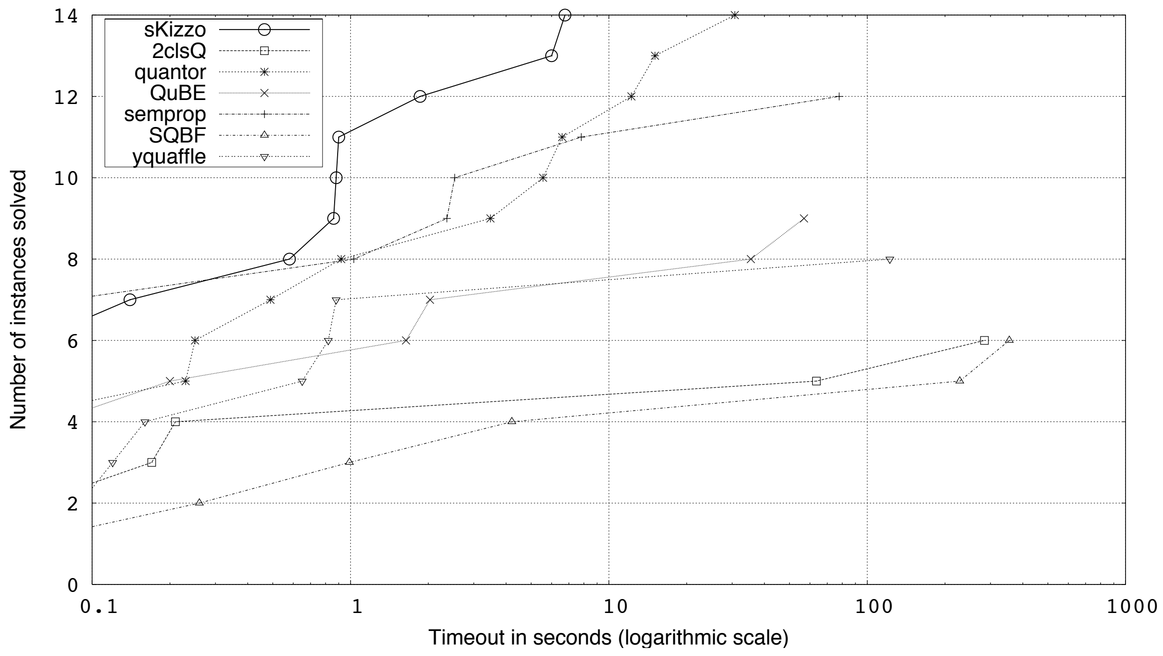


Figure 5. Performance of several QBF solvers over Ling's and Pan's FV benchmarks (14 instances).

reasoning in the symbolic space may occasionally lead to more and more complex symbolic representations for a smaller and smaller propositional expansion. In particular, the discovery of symbolic equivalences may trigger the application of substitutions that decrease the size of the propositional expansion while increasing the number of symbolic clauses necessary to represent such expansion. In rare cases, such behavior generates an overwhelming number of symbolic clauses. To avoid such problem, we disabled SER. sKizzo then solved the 4 smaller instances (25%) in approximately 15 seconds overall, and timed-out again on the other cases. The new inference logs reveal, surprisingly, that the solver is using search (B style), after both Q and S quickly reach their fixpoints with no verdict. Expansion to SAT stays always unaffordable. Given that (i) the search-based engine is the weakest part of sKizzo (due to more general, hence less efficient, data-structures than plain search requires) but (ii) no search-based solver answered, we argue that it is indeed beneficial to preprocess by Q and S before searching.

Lahiri/Seshia. No solver is able to evaluate any of these three instances. To the best of our knowledge, they have never been solved in their original, non-preprocessed form[7]. The analysis of our solver's trace suggests that the most effective inference attack to such instances is the one described below for the Mneimneh/Sakallah case, as the BDD package gets stuck in the attempt to suitably reorder the universal variables.

7. It has been recently reported [72] that two of the three instances in the *uclid* family have been solved by using sKizzo in combination with the preprocessor preQuel.

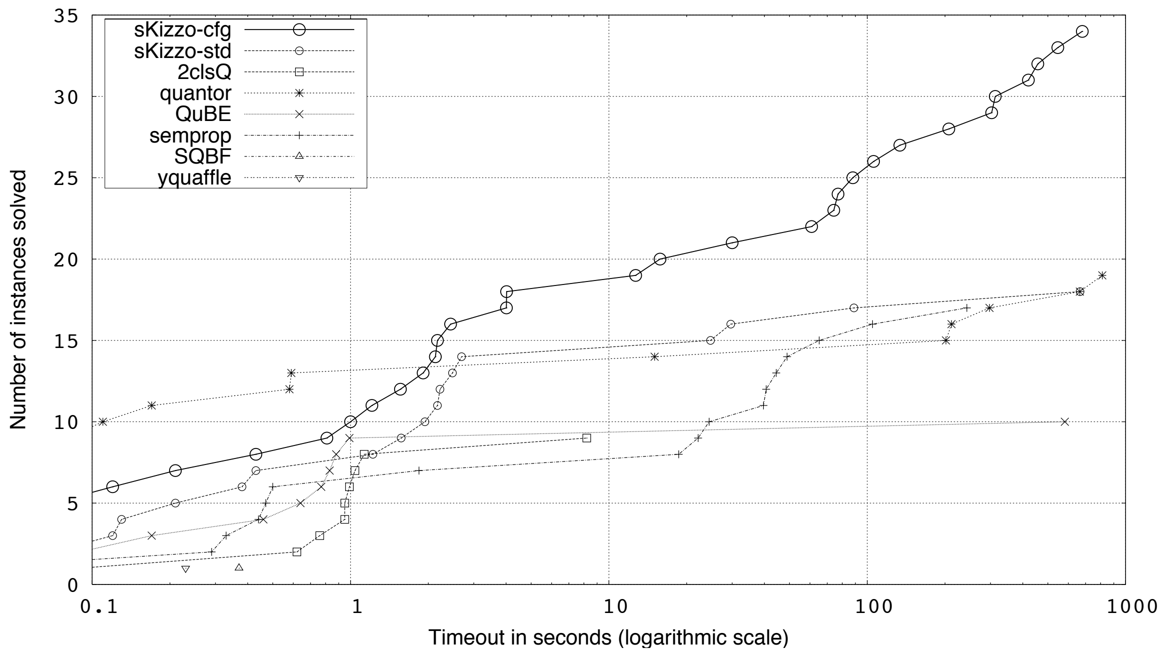


Figure 6. Performance of several QBF solvers over Mneimneh-Sakallah's benchmarks (90 inst.).

Ling and Pan. These families are examined together, as they contain relatively small and easy instances. Both sKizzo and quantor decided the whole benchmark set (the former within a few seconds) while none of the search-based reasoners manages to achieve the same result. A cumulative comparison for both families is reported in Figure 5.

Mneimneh/Sakallah. All the solvers show similar performances on these families, with the best one (quantor) reaching a modest 21% (Figure 6). To explain why sKizzo's performance is so close to that of search-based solvers, we first hypothesized that search is indeed the inference style it adopts. However, the analysis of the inference logs reveals that all the time is spent in the S state, with the bottleneck lying in the interaction between symbolic unit clause propagation, equivalence reasoning, and BDD dynamic reordering. This circumstance gives us the occasion to exemplify one peculiar feature of sKizzo, related to its highly modular architecture: The large degrees of freedom in its configuration, unparalleled in simpler architectures. Once properly tuned on the instances at hand, the solver can in general do (much) more than it does in the standard configuration. We test an alternative configuration which requires to avoid BDD reordering altogether (usually automatic reordering helps performance, so it is on by default) and to apply simplification by equivalence reasoning only after SUCP reaches its fixpoint. The results get substantially better (sKizzo-cfg in Figure 6, as opposed to the solver in its standard configuration, labeled by sKizzo-std).

The improvement we obtain is remarkable for three reasons. First, these instances are a significant fraction of all the formal verification instances in the QBFLIB. Second, they encode real-world verification problems on circuits using a well known method [59], thus being an important reference point. Third, they show the worst (smallest) solved/unsolved ratio among all the families in the QBFLIB, apart from the Lahiri/Seshia instances.

Interestingly, in the second configuration, sKizzo leverages all its inference engines but search, according to a pattern which comes out to be the same across all the instances in each family, and across every family in the group. First, it *normalizes* the instance with an aggressive application of the SUCP rule. This operation takes more than 90% of the running time. When SUCP reaches its fixpoint, the symbolic size of the formula is sensibly smaller than it was originally, but its ground projection—though greatly reduced—stays definitely unaffordable (order of 10^{100} clauses). After SUCP finishes, SER comes into play and succeeds in inferring certain equivalences that allow the substitution of many "lightly" quantified existential variables (few dominating universal quantifiers) for other "heavily" quantified existential variables (many dominating universal quantifiers). These substitutions are beneficial, in that the migration (via substitution) of literals towards less deep scopes substantially shrinks the ground projection of the clauses they belong to. For example, consider a clause $a \vee b$ under the quantification $\exists a \exists x \forall Y \exists b$, where Y is a set of universal variables and x is a variable in the same existential scope as a. The propositional expansion of $a \vee b$ may yield exponentially many clauses in the size of Y. However, if x is proved to be equivalent to b and substituted to it, the expansion of $a \vee b$ becomes independent of the size of Y. In particular, if no other universal variable dominates a and x, the expansion has size 1. A few of these substitutions can suddenly (and exponentially) compress the propositional expansion of the formula, making it tractable. This is what happens in all the Mneimneh/Sakallah benchmarks.

The ground expansions are all SAT-solved in a matter of seconds. The key towards solving even more instances in this group seems to lie in a better interplay between the heuristics used to schedule symbolic units during SUCP, and the heuristics used to reorder BDDs.

Scholl/Becker. The comparison, reported in Figure 7, shows that most solvers converge to decide slightly more than 50% of the instances at extreme timeouts, with no one showing any real edge over the others. The shape of the curves suggests the presence of many easy cases for sKizzo and quantor (decided within one second), followed by much harder ones. Conversely, search-based solvers scale more smoothly, even though quantor is solving overall the largest number of instances. The analysis of the inference log of sKizzo reveals that it is spending most of the time in the B state (after symbolic simplifications terminate, and after resolution is arrested for the number of resolvents starts diverging). In this benchmark, sKizzo matches the performance of search-based solvers using search itself. However, its branching engine has largely sub-optimal raw efficiency due to the use of complex data structures, so we think previous simplifications (achieved in the S state) are what re-equilibrates the performance. Given that such simplifications make it possible for the solver to match the performance of search-based competitors while using a weaker search-engine, we expect that large room for improvement lies in applying a better branching engine after S terminates.

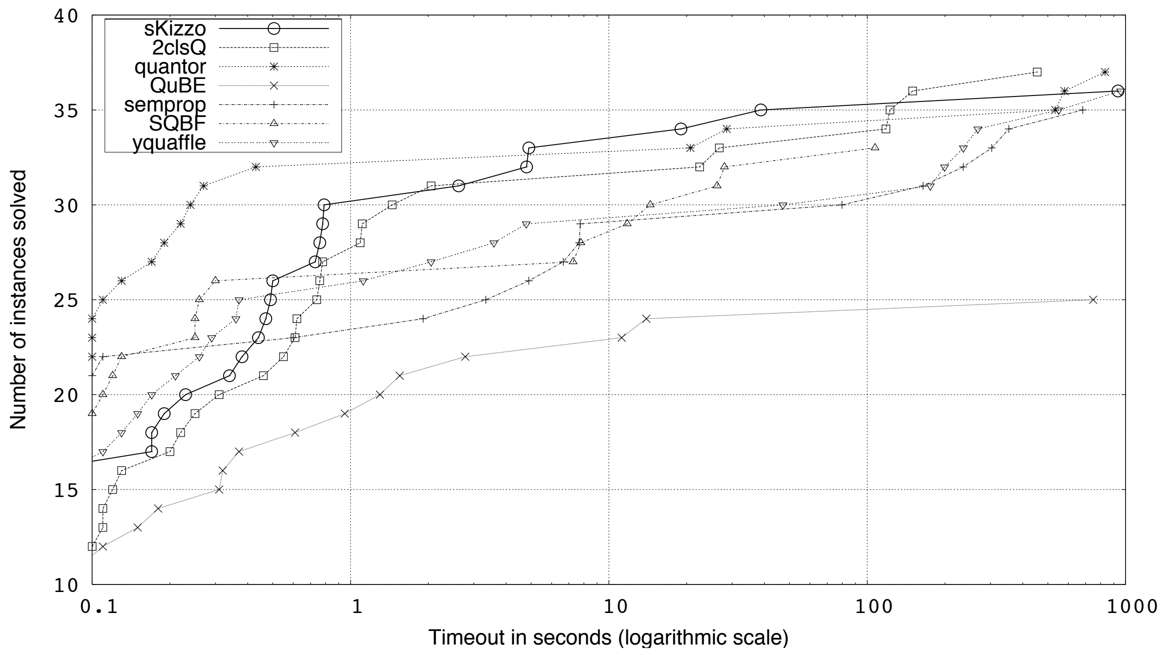


Figure 7. Performance of several QBF solvers over Scholl-Becker's benchmarks (64 instances).

2.6 QBF-Based versus SAT-Based FV

Just like SAT solvers a few years ago, QBF solvers are exhibiting a promising performance escalation these days. However, irrespective of the relative merits of solvers, long-term meaningful accomplishments for QBF-based FV have to be measured against alternative approaches to the same tasks. Not surprisingly, the first competitor of QBF in formal verification is SAT. On one hand, SAT-based encodings lend by their nature to a fair comparison. On the other hand, QBF heavily compresses the encoding, and this leads us to wonder whether gains can be expected at runtime.

Unfortunately, the SAT arena has turned out to be quite unfavorable to QBF. All the experimental comparisons carried out recently yield (extremely) negative results [75, 7, 59, 52, 20, 63, 51, 46, 45].

For example, Mneimneh and Sakallah consider in [59] the QBF formulation of the vertex eccentricity problem. They devise an improved and simplified version of the formula presented in Section 2.2, and experiment with the latter. The experimental analysis shows, in their own words, that "*the reduced formulas are unsolvable by state-of-the-art QBF solvers*[8]." and that the "*SAT-based solution outperforms state-of-the-art QBF solvers*".

Biere in [20] compares (experimentally) QBF-based BMC against SAT-based BMC over different models of counters. The results of this evaluation suggest that the QBF approach to MC may *at most* match the results obtained by plain BMC. Moreover, as the examples analyzed in [20] are very easy for classical BDD-based model checking, the author con-

8. At the time the paper we are discussing was written, only search-based QBF solvers were available.

cludes that some kind of remarkable improvement is expected from alternative QBF solver architectures.

Katz, Hanna and Dershowitz [47, 35, 46] report even more (negatively) impressive results. They compare SAT-based and QBF-based MC over thirteen proprietary real-world test cases comprised of 234 instances. SAT solvers conquer 78% of these instances, while general purpose QBF solvers stop at 1%. In the QBF solver evaluation described in [51], the FV families of instances are those contributing the most to the set of unsolved instances. For example, no more than 20% of the seven-year old QBF benchmarks over the ISCAS89 circuits have been solved.

In [45], Jussila and Biere present a framework allowing for SAT/QBF comparisons on a broad set of FV tasks. They describe a tool to transform *SMV specifications* (a standard input language for model checkers) into propositional encodings, over which all the MC tasks introduced in Section 2.2 (and others) can be performed via QBF. They report experimental results for many variants of BMC and k-induction encodings. After observing, experimentally, a significant compression in the size of the instances, they conclude by saying that "*QBF solvers seem not to be able to take advantage of these compact encodings*", and "*much more research in QBF is needed to be able to use QBF as alternative to SAT-based MC [...]*".

3. QBF in FV: Promising Perspectives

Many research efforts aiming to ameliorate the disappointing situation delineated in Section 2.6 are going on. In this section, we first introduce and discuss a list of keywords which can be used to roughly categorize promising research perspectives. Then, we detail some of these possibilities.

More efficient implementations. As SAT solvers demonstrated, implementing things "the right way" in propositional reasoners may have a huge impact on raw performance. The quest for small but effective breakthroughs on this side is an ever-ongoing effort, fostered by international events such as the QBF competition track (held during the SAT series of conferences).

Alternative decision paradigms. Both quantor and sKizzo radically depart from classical search-based approaches. Their unexpected efficiency suggests that decision paradigms other than search may have more chances to succeed in QBF than they had in SAT. The interplay between universal and existential quantification seems indeed to offer room for a broader variety of solver architectures. As mentioned in Section 2.4, decision procedures applying BDDs to support either inference or search have been presented recently [63, 6, 40], as well as algorithms that combine SAT solvers with QBF solvers [8, 11, 70]. Furthermore, entirely novel approaches exist, such as the mixed search/inference algorithm outlined in [17]: The search branches on existential variables but not on universal ones, whose admissible values are inferred. Another promising possibility is that of *preprocessing*. In [72] a preprocessing method based on binary clauses is presented which is shown to enhance the performance of all the QBF solvers to which it has been applied. Finally, attempts to guide search-based solvers towards fragments that can be recognized and decided in polynomial time seem promising [68].

We do not further elaborate on the above topics, yet a section is devoted to each of the following.

Beneficial language/solver extensions. As first identified in [5], quantified conjunctive languages like QBF (or, QCSP [22]) may have *intrinsic* difficulties in modeling some ubiquitous features of the scenarios they should target. Evidence has been provided [5, 19] that not only is modeling difficult, but that solvers may find it artificially hard to tackle the "contorted" encodings used as a workaround to sidestep modeling hindrances. In search-based solvers this effect is known as the "illegal search space issue" [5], i.e., the time-consuming exploration of large parts of the search tree, where "obviously" no solution exists, as an awkward side effect of the techniques used to model certain concepts. Solution attempts include the use of "indicator variables" [5], and the employment of combined DNF/CNF representations [83, 69]. Other general solutions are possible, in the form of pure *language extensions* not specific to search-based solvers—a positive attribute given the relative performance of search-based and alternative solvers on FV instances (Section 2.5). For example, AIGs (And-Inverter Graphs [49, 2]) could be used to capture quantified models. Or, QBFs with *restricted quantification* may be employed. Restricted quantification [19] give means to confine the span of quantification to specific combinations of assignments (rather than unselectively to all of them), and this eases considerably the construction of many QBF encodings (and, hopefully, their solution). Such quantifiers require neither DNFs nor additional auxiliary variables, are general enough to help in other frameworks (such as QCSP), and are not specifically designed for search-based approaches. We discuss this extension in Section 3.1.

Alternative problem encodings. While a lot of effort has been devoted to design new decision procedures for QBF, and to implement them, much less care has been reserved to the other side of the coin, i.e., to investigate "the right way" of encoding problems. Classic QBF-based formulations of FV tasks (Section 2.2) are both intuitive and quite compact. However, empirical evidence suggests that they yield instances not easily solvable with current technology (Section 2.6). One question thus naturally arises: Is it conceivable to explore alternative encoding techniques, or perhaps radically new ones, aiming to improve the response of solvers? A tentative but very encouraging answer to this question is advanced in Section 3.2.

Exploitation of specific QBF features. While QBF formulas share a lot with SAT instances, they also differ in many respects from their "purely existential" relatives. Most differences appear at first as just "negative" features, but they might later disclose a hidden potential. For example, the basic task of verifying the validity of a (true) QBF is in general intractable [27], and the very exercise of extracting and representing a witness of validity is not trivial. However, once the theory and technology to achieve this result exist [13], applications to real-world scenarios may quickly materialize (see for example [79]). Another intriguing possibility is that of QBF formulas where some variables are neither universally nor existentially quantified (i.e., they stay free). While all present solvers just consider closed QBF instances, the role of open QBF and the possibilities they disclose in FV deserve further attention. We examine the former topic (the role of validity certification) in Section 3.3, and the latter (open QBF instances) in Section 3.4.

3.1 Restricted Quantification

The most intuitive way to make sense of a QBF is to think of it as a *game* between two players. One player, called ∃-player, is associated with the existential quantifier, the other one, called ∀-player, is associated with the universal quantifier. The goal of the ∃-player is to satisfy all the clauses, hence the matrix as a whole. The goal of the ∀-player is to violate at least one clause, thus overcoming the opponent's effort. The two players play against each other in turn, for a finite and fixed number of rounds. The moves they do consist of assigning truth values to variables. Which variables get assigned at each step is statically decided by the left-to-right precedence order given in the prefix.

3.1.1 The Problem

One manifest problem in such QBF-based modeling is the impossibility to cleanly restrict what the universal player is allowed to do. The complication, in a nutshell, is the following: With games, almost invariably come *rules*. Rules preclude some choices as a function of previous moves by the same player or by the opponent. An elementary example is the prohibition in most board games to play in a cell already occupied by someone. In general, rules are arbitrarily complex patterns of forbidden assignments which dynamically restrict the set of legal moves over a game life-span, in so as to comply with an underlying *game discipline*. The observance of such a discipline is what purely conjunctive languages like QBF with CNF matrixes (or QCSP) find difficult to enforce.

In principle, both players can be ruled the same way: It is a matter of stating that if a player chooses a forbidden move, he loses. Such a threat is promptly posed to the ∃-player. We consider the membership of the move to the set of legal moves as just an additional constraint. If the ∃-player cheats, he falsifies this additional constraint and hence loses the game. No similar expedient can be used against the ∀-player: The game is a loss for him when all the clauses are satisfied, something which simply cannot be imposed by just *conjuncting* whatever additional constraint.

A straightforward workaround exists though: Modify the whole set of clauses in such a way that they are "automatically" satisfied by any cheating attempt of the universal player [9]. One clean way to do this is to first express the game and its rules, including the ∀-player discipline, by some non-CNF formula, which is then transformed into CNF with the help of (innermost existentially quantified) auxiliary variables [65]. This trick has been widely adopted—often tacitly—in QBF modeling, and is one of the reasons why an entire

9. More precisely, by cheating attempts *not preceded by illegal moves of the existential player.* What matters is indeed *which player cheats first.* For example, if ∀-player cheats after an illegal move of ∃-player already occurred, the whole matrix has to reveal a contradiction. Conversely, the matrix has to be satisfied even if the existential player cheats, provided the ∀-player committed some previous infraction. The information contained in the prefix on the alternations of scopes is of key importance to determine the outcome of those games in which both players may cheat.

library of real-world instances exists [43]. The formalizations presented in Section 2.2, for example, use a non-CNF syntax and are supposed to go through a "CNF-ization" process before being fed to standard QBF solvers. Tools entirely devoted to automate the translation of non-conjunctive quantified formulas into a conjunctive form exist (e.g., `qst` [85]). Unfortunately, such translation may "confuse" QBF reasoners and lead them to do more work than necessary, as thoroughly discussed in [5].

3.1.2 A Radical Solution

A complete solution to this problem can be obtained by porting to QBF the notion of *restricted quantification*, recently implemented in QCSP solvers [19]. Rephrased in the (conjunctive prenex) QBF framework, a restricted quantifier $\mathsf{q}X[L]$, with $\mathsf{q} \in \{\exists, \forall\}$, is a quantifier that considers all the combinations of assignments to the set of Boolean variables X, yet only spans over those assignments which satisfy the *restriction* L, expressed as a CNF. Let us call QBF$^+$ the QBF language extended with restricted quantification. For example, in QBF$^+$ the formula

$$\forall x_1[L_1^{\forall}(x_1)] \;\; \exists y_1[L_1^{\exists}(x_1, y_1)] \;\; \forall x_2[L_2^{\forall}(x_1, y_1, x_2)] \;\; C(x_1, y_2, x_2) \tag{15}$$

reads "*for all the assignments to x_1 such that $L_1^{\forall}$, there exists an assignment to y_1 such that $L_1^{\exists}$ and for all the assignments to x_2 such that $L_2^{\forall}$, C is satisfied*". The legal opening moves for $\forall$-player are thus constrained to be models of the CNF $L_1^{\forall}(x_1)$. Next, $\exists$-player's reply is constrained by $L_1^{\exists}(x_1, y_1)$: Once the choices over x_1 from $\forall$-player's side are known, an assigment over y_1 is to be considered only if it is a model of $L_1^{\exists}$. Likewise, the restriction $L_2^{\forall}(x_1, y_1, x_2)$ is provided.

It is not difficult to reshape (15), or any other QBF$^+$ formula, as a standard non-prenex QBF: In spelling out the meaning of our sample QBF$^+$ we used three times the "such that" connective to introduce the restrictions quantifiers are subject to. The "such that" connective stands for a *conjunction* when it relates to moves of the $\exists$-player, and for an *implication* when $\forall$-player is concerned.

Formally, let F be a QBF$^+$ formula made up of a sequence of restricted quantifiers $\mathsf{q}_1 X_1[L_1] \cdots \mathsf{q}_n X_n[L_n]$ followed by a propositional formula C built on the variables $X_1, \ldots, X_n$. Let us write $F = \mathsf{q}X[L] \,|\, F'$ to mean that $\mathsf{q}X[L]$ is the first restricted quantifier in the sequence of quantifiers of the QBF$^+$ F, while F' is the formula obtained from F by removing such first quantifier. Then, a translation from QBF$^+$ into (non-conjunctive) QBF which formalizes restricted quantification is:

$$Tr(F) = \begin{cases} \exists X(L \wedge Tr(F')) & \text{if } \; F = \exists X[L] \,|\, F' \\ \forall X(L \rightarrow Tr(F')) & \text{if } \; F = \forall X[L] \,|\, F' \\ F \quad \text{otherwise} & \end{cases} \tag{16}$$

For example, the translation of the sample QBF$^+$ (15) is:

$$\forall x_1(L_1^{\forall}(x_1) \rightarrow \exists y_1(L_1^{\exists}(x_1, y_1) \wedge \forall x_2(L_2^{\forall}(x_1, y_1, x_2) \rightarrow C(x_1, y_1, x_2))))$$

By rewriting implications as disjunctions and by pushing quantifiers outwards we obtain the equivalent prenex form:

$$\forall x_1 \exists y_1 \forall x_2. \;\; (\neg L_1^{\forall}(x_1) \vee (L_1^{\exists}(x_1, y_1) \wedge (\neg L_2^{\forall}(x_1, y_1, x_1) \vee C(x_1, y_1, x_2)))) \tag{17}$$

The latter rewrite can be solved by either employing a non-CNF solver for QBF, or by converting its matrix into CNF. The conversion to CNF enables the use of standard QBF solvers, but it obfuscates the structure of the original problem, thus causing the "illegal search space" and related issues [5]. The solution of (17) via non-CNF solvers is a more intriguing possibility, yet arguments in favor of a direct language extension to the form (15) exist:

1. The disjunctions and negations in (17) are *not* used to capture relevant facts or rules of the game (to this end, the conjunctive formulas L_i^q and C suffice). Rather, they arise from the disjunctive meaning of the "such that" particle after universal quantification. So, it is not from modeling the game but from modeling *the question we pose on the game* that the non-conjunctive form originates. Furthermore, the common origin of disjunctions and negations makes the syntactic structure of (17) very regular and predictable. Rather than extending the reasoning engine towards general non-conjunctive languages, it could be convenient to design a language in which disjunctive semantics is confined to special points only—as in (15)—and then develop a specialized mechanism to efficiently handle the resulting quantified structure.

2. Data structures and algorithms to deal with CNFs are now mature and extremely efficient. Most breakthroughs in propositional reasoners—such as lazy data structures, fast unit clause propagation techniques, efficient learning schemes—target CNF formulas[10.]. The special form of (15) allows the reuse all this background, because the formulas L_i^q and C are in conjunctive normal form. Such clause "containers" can, for instance, be represented by watched data structures, and any local form of inference, e.g., unit propagation, can reuse existing fast algorithms. Of course, some additional mechanism is necessary to deal with the sequence of alternated restricted quantifiers, but the bulk of the reasoning task is in handling CNFs.

3. As years of successful SAT-based modeling prove, most concepts are naturally captured as conjunctions of constraints. So, the syntax (15) is not only amenable to be decided by adapting existing technology and algorithms, but it is also cognitively adequate to represent game-with-rules scenarios from the *modeler's viewpoint*. In essence, we argue that the modeler does not need to use much non-clausal structures once restricted quantifiers are available.

4. Restricted quantifiers are not meant to be dealt with by search-based solvers only (while most recent extensions to QBF, e.g., [5, 83, 69, 70, 68], exclusively target this class of solvers). Restricted quantifiers come in the form of a pure language extension, not tailored to the details of any specific inference strategy. So, it is possible to adapt a DPLL-like QBF engine to understand restricted quantifiers just like it is possible to extend other reasoning frameworks. This is a good feature, as many experiments in this paper suggest that search-based QBF solvers are not necessarily the best option so long as FV applications are concerned.

10. Some of these techniques have then been adapted to non-clausal frameworks; see e.g., [82].

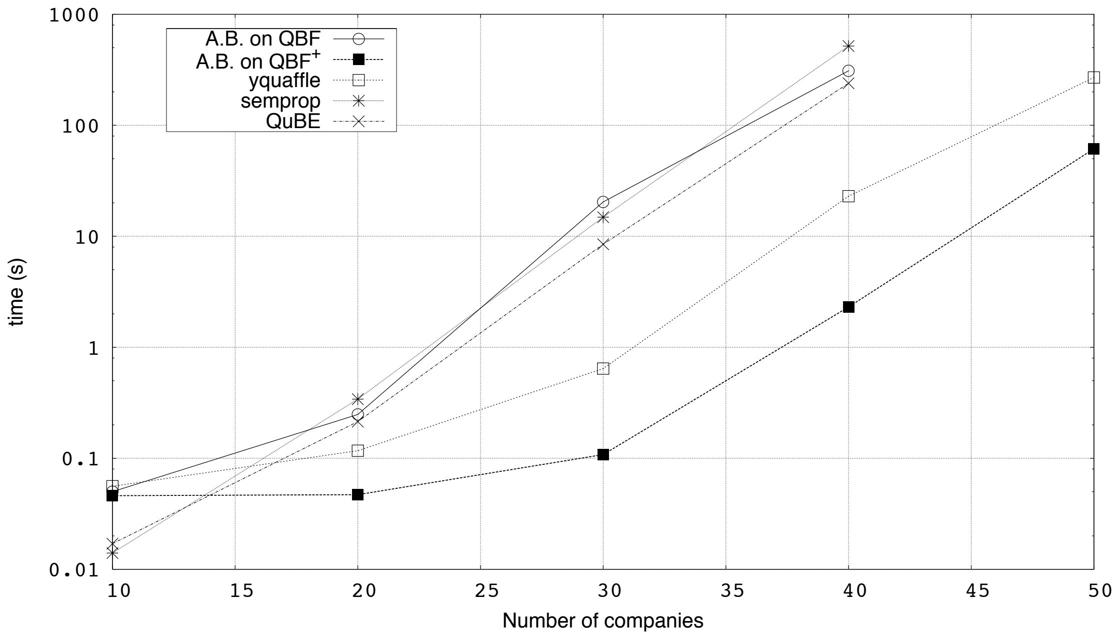


Figure 8. The impact of restricted quantification: Run-time comparison over the "Strategic Company" example. The X axis gives the number of companies in the set. The Y axis gives run-time averaged over 100 randomly generated instances for each size of the set of companies.

3.1.3 A Simple Case Study

To exemplify the arguments discussed in the previous section, we consider a common modeling problem, which arises in many practical applications.

Example 2 *Suppose we have a set of objects C of size n, and suppose some of its subsets $S \subseteq C$ enjoy a certain property P. The problem is to decide whether an element $x \in C$ is "important" in the sense that it belongs to at least some* minimal *subset of C with the property P. In propositional logic, this scenario can be captured by using a set S of n variables to select any subset of C, and a CNF on the variables S to express the property of interest, written $P(S)$. By means of restricted quantification, this problem is readily formalized. An element x is not important if every time it belongs to some subset S with the property P, a subset $S' \subset S$ preserving P exists in which x does not appear:*

$$\forall S[S \subseteq C \wedge P(S) \wedge x \in S] \;\; \exists S'[P(S') \wedge S' \subset S]. \;\; x \notin S' \tag{18}$$

Or, equivalently:

$$\forall S \; \exists S'. \;\; [S \subseteq C \wedge P(S) \wedge x \in S] \rightarrow [P(S') \wedge x \notin S' \wedge S' \subset S] \tag{19}$$

We extended the non-standard QBF algorithm [17] to deal with $\forall\exists$ formulas with restricted quantification[11], and we fed it with formulation (18). For comparison, a prenex CNF

11. The algorithm is called "abstract branching", or A.B. for short. Both the original algorithm and its version extended to QBF$^+$ are implemented inside sKizzo.

conversion of (19) has been provided as input to standard QBF solvers. The property P used in our experiments is the "strategicity", taken from [28] and defined as follows. C is a set of companies, each one producing some goods but not others. A subset $S \subseteq C$ has the property P if the companies it contains produce overall the same set of goods produced by C, i.e., the subset of companies covers all the production portfolio. A company is strategic if it belongs to at least one minimal portfolio-preserving set.

Figure 8 presents the result of the comparison. Even if the original solver (A.B. on QBF) was not the best one on this family, its version extended with restricted quantifiers (A.B. on QBF$^+$) improves by two orders of magnitude and outperforms other approaches.

3.1.4 Restricted Quantifiers in FV

The impact restricted quantifiers may have on QBF-based FV tasks is not clear. On one hand, most of the arguments used in, e.g., [69, 5, 19] to expose the weaknesses of purely-conjunctive QBFs rely on the presence of many quantifier alternations, while most FV applications generate instances with a fixed and small number of alternations. On the other hand, the "strategic company" example suggests that even the basic one-alternation case may benefit from restricted quantification.

In any case, it is important to note that "game-with-rules" scenarios (via restricted quantification) are not just artificial, or unrealistic examples. They seem to satisfy the most ubiquitous and inherent needs of any act of modeling operated through mixed universal/existential quantification. For instance, QBF formulations of classical FV problems can be lifted to QBF$^+$ quite smoothly.

Example 3 *The QBF$^+$ equivalent of the BMC formulation (4) from Section 2.2 is*

$$\exists s_0, \ldots, s_k\ [I(s_0) \wedge B(s_k)] \quad \forall x, x'\ [\vee_{i=0}^{i=k-1}(x = s_i) \wedge (x' = s_{i+1})]. \quad T(x, x')$$

which reads as a natural definition of a (bad) path: "a sequence of states starting in one initial state and ending in a bad state, such that for any two adjacent states x, x' along such sequence we have that the transition from x to x' is valid". Note that inside every restriction and in the final formula there are only conjunctions and formulas classically written in CNF, such as $T(x, x')$.

Example 4 *The QBF$^+$ formulation of (6) from Section 2.2 may be written as*

$$\forall \alpha_1, \ldots, \alpha_k\ [one(\alpha_1, \ldots, \alpha_k)] \quad \exists x, x'\ [\wedge_{i=1}^{i=k}(\neg\alpha_i \vee (x = s_{i-1} \wedge x' = s_i))]. \quad T(x, x')$$

Note how in this case the one function and restricted quantification go hand in hand to characterize a special universal quantifier which is meant to consider all and only the combinations of assignments to $\alpha_1, \ldots, \alpha_k$ where exactly one variable is true.

Example 5 *The formula (10) of Section 2.2 for eccentricity computation becomes*

$$\exists s_0, \ldots, s_k\ [P^k(s_0, \ldots, s_k)] \quad \forall t_0, \ldots, t_{k-1}[P^{k-1}(t_0, \ldots, t_{k-1})]. \quad \wedge_{i=0}^{k-1} \neg(s_k = t_i)$$

whose interpretation closely mimics the definition of (lower bound for the) diameter: "there exists a valid path of length k with last state s_k such that every valid path of length $k-1$ fails to reach s_k".

Example 6 *The characterization of simple paths (11) in Section 2.2 is rephrased as*

$$\forall \alpha_0, \ldots, \alpha_k\ [one(\alpha_0, \ldots, \alpha_k)] \quad \exists x.\ \wedge_{i=0}^{i=k} (\alpha_i \leftrightarrow (x = s_i))$$

where the first quantifier is restricted, while the second is not.

Example 7 *The iterative application of the non-copying squaring rule from Section 2.2.3 generates a non-prenex non-CNF formula which nests an alternation of universal/existential quantifications and conjunctive/disjunctive connectives. This happens to be precisely the syntactic form captured by restricted quantifiers. Let us use the shorthand "$\langle s, s'\rangle m\langle x, x'\rangle$" to mean "$(x = s \wedge x' = m) \vee (x = m \wedge x' = s')$". Then, n applications of the rule (8), rephrased in QBF$^+$ and applied to express the existence of a valid path between an initial state s_i and a bad state s_b, produce:*

$$\begin{array}{l}
\exists s_i[I(s_i)]\ \exists s_b[B(s_b)] \\
\qquad \exists x_1 \forall x_2, x_3[\langle s_i, s_f\rangle x_1\langle x_2, x_3\rangle] \\
\qquad\qquad \exists x_4 \forall x_5, x_6[\langle x_2, x_3\rangle x_4\langle x_5, x_6\rangle] \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad \exists x_n \forall x_{n+1}, x_{n+2}[\langle x_{n-2}, x_{n-1}\rangle x_n\langle x_{n+1}, x_{n+2}\rangle].\ T(x_{n+1}, x_{n+2})
\end{array}$$

This formula characterizes the existence of bad paths of length 2^n using n alternations, $2n$ scopes, and $2 + n$ restricted quantifiers. Its matrix just consists of (one copy of) the transition relation.

In all these examples, the equivalence to the original formulation is easy to prove by applying the translation (16). More than the proof of equivalence, what is interesting to observe is how all these different FV encodings are amenable to be cast in QBF$^+$, and produce a natural (and, hopefully, easier to solve) formulation. We are currently assessing how much solvers can profit from these more structured formalizations, working specifically on skolemization-based solvers.

3.2 Alternative Encodings

Most attempts to use QBF in FV share a "single point of failure": They all rely on the same encoding schemes. For example, in BMC, the ones described in Section 2.2 and small variants thereof.

Interestingly, recent results [55, 56] suggest that alternative encodings can indeed make a difference. We present here a simplified account of a new QBF-based BMC technique, which improves on a couple of significant ways over classical schemes. The reader is referred to [55, 56] for details.

As usual, a safety property is to be checked within a bounded horizon. Instead of a single copy of the transition relation, w copies are used (where $w > 1$ is a fixed parameter taking small values). These copies are interconnected through a partial explicit unrolling over consecutive states:

$$T_w(s, s') \equiv \exists x_1, \ldots, x_{w-1}.\ T(s, x_1) \wedge T(x_1, x_2) \wedge \cdots \wedge T(x_{w-1}, s') \tag{20}$$

The formula $T_w(s,s')$ characterizes w-step long valid paths, i.e., it is $T_w(s,s')$ for all the states s' that can be reached from s in w steps, traversing some intermediate states $x_1,\ldots,x_{w-1}$.

Let us call *frame* each w-step long path characterized by $T_w(s,s')$, and let us consider a model checking bound k such that $\frac{k}{w}=2^n$ (this is a simplifying assumption, relaxed in [55, 56]). We can decompose any valid path of length k into 2^n *contiguous* frames. Conversely, we can characterize paths of length k by securing the contiguity of 2^n w-step frames like meshes in a chain. Securing the contiguity of frames is just like encoding the BMC problem for a system whose transition relation is $T_w(s,s')$ (instead of $T(s,s')$). So, we could use the iterative squaring or the linear selection technique from Section 2.2. We use instead a new method, based on two coupled *multiplexers* with a common n-bit selector vector. The behavior of an n-bit multiplexer with inputs $s_0,\ldots,s_{2^n-1}$ and output s can be declaratively characterized as

$$mux_n(s,t,s_0,\ldots,s_{2^n-1}) \equiv (s = s_{\lambda(t)}) \tag{21}$$

where $0 \le \lambda(t) < 2^n$ is the decimal value of the integer logarithmically encoded in the n-bit selection vector t. Of course, a direct and effective "implementation" in Boolean logic (such as the one described in [55]) is required for the technique to work in practice, but for the sake of our argument we are content with (21). By coupling the selectors of two multiplexers, as in $mux_n(s,t,s_0,\ldots,s_{2^n-1}) \wedge mux_n(s',t,s'_0,\ldots,s'_{2^n-1})$, we can constrain the couple of states (s,s') to equal the couple of states $(s_{\lambda(t)},{s'}_{\lambda(t)})$ for any desired $\lambda(t)$. If we connect the two multiplexers to the same inputs $s_0,\ldots,s_{2^n}$ in a 1-step displaced fashion, as in $mux_n(s,t,s_0,\ldots,s_{2^n-1}) \wedge mux_n(s',t,s_1,\ldots,s_{2^n})$, we make the couple (s,s') equal the consecutive states $(s_{\lambda(t)},s_{\lambda(t)+1})$. By quantifying universally the bits of the shared selector t, we cover all the couples of consecutive states, so that a single constraint $T_w(s,s')$ can be used to enforce the contiguity of all the frames:

$$\forall t\ \exists s,s'.\ [mux_n(s,t,s_0,\ldots,s_{2^n-1}) \wedge mux_n(s',t,s_1,\ldots,s_{2^n}) \wedge T_w(s,s')] \tag{22}$$

This formula thus describes a path of $w \cdot 2^n$ transitions organized in 2^n contiguous w-step frames, and it can be used to perform BMC. The states $s_0,\ldots,s_{2^n}$ are traversed by the system after $0,w,\ldots,2^n\cdot w$ steps respectively. Intermediate states are "hidden" in the existential variables quantified in the third scope, according to (20). The existence of a 2^k-step bad path is captured as:

$$\begin{aligned}\exists s_0,\ldots,s_{2^n}\ I(s_0)\ \wedge B(s_{2^k}) \wedge&\\ \wedge\ \forall t\ \exists s,s'.\ [mux_n(s,t,s_0,\ldots,s_{2^n-1}) \wedge mux_n(s',t,s_1,\ldots,s_{2^n}) \wedge T_w(s,s')]&\end{aligned} \tag{23}$$

The rationale behind this alternative encoding is as follows. On one hand, the (limited) explicit unrolling of the transition relation performed in (20) enhances the solver's strength at performing direct inferences which span a few contiguous states. The solver thus reasons on a model in which the underlying short-term logic is brought out into the open, rather than fragmented in minimal segments by the intervention of quantifiers. On the other hand, (22) preserves the advantage of the iterative squaring method (number of universals

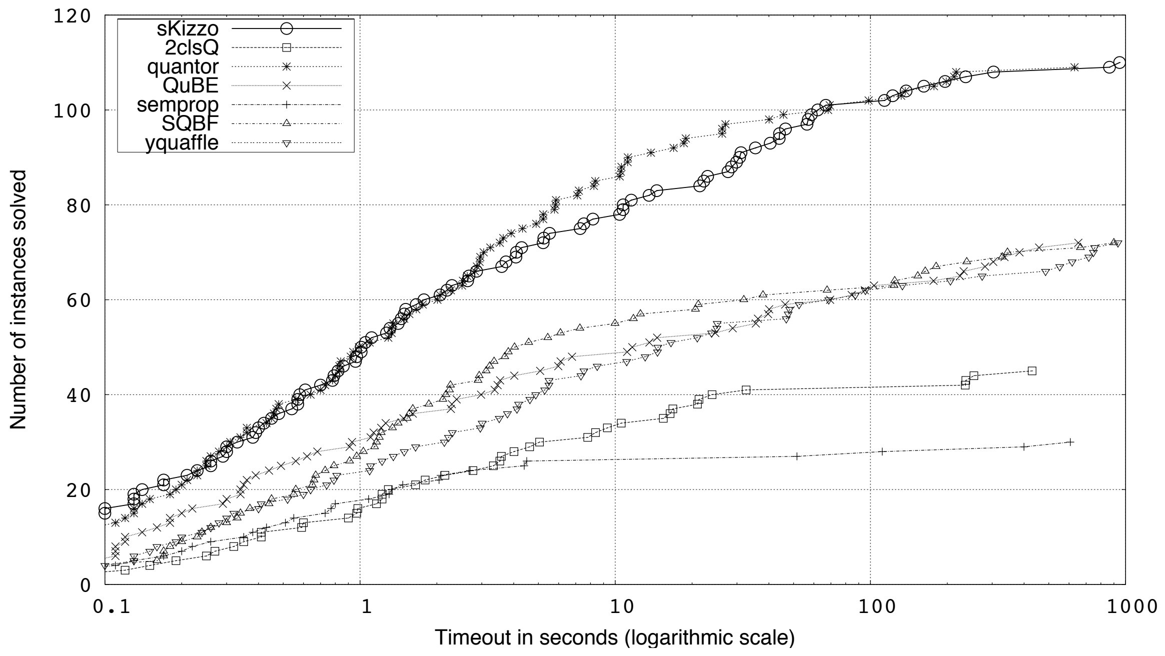


Figure 9. Overall performance of several QBF solvers over BMC instances from six designs of real-world circuits (from opencores [1]), encoded according to the technique described in [56].

logarithmic in the bound) while operating with a fixed number of quantifier alternations and a linear representation of time, as in (4–6).

Another interesting property of the new method is that the size of the encoding (23) can be *minimized* by optimizing the length of frames. For a fixed bound k, the longer the frames (i.e., the greater the value of w) the smaller the number of frames we need to connect. However, increasing the length of frames has a non-monotonic effect on the size of (23): Longer frames demand more explicit copies of the transition relation, according to (20), thus increasing the size of (23). At the same time, the longer the frames, the smaller the number of joints among frames. Less joints means smaller multiplexers, i.e., smaller encodings for (21). The size of (23) as a function of w can be determined exactly (though this involves some details, see [56]) and minimized numerically.

Figure 9 shows how different QBF solvers react to this new encoding on some real-world designs of circuits (such designs have been downloaded from the repository opencores [1], and their QBF encodings have been contributed to the QBFLIB archive [42]). Once again, search-based solvers are not competitive with alternative ones. sKizzo and quantor performances are significantly better than the best search-based solver (with quantor prevailing on sKizzo for timeouts between roughly 2 seconds and 1 minute). The two performance profiles are surprisingly similar, considering that the two solvers are behaving in completely different ways: quantor, as usual, proceeds by quantifier elimination; sKizzo is simplifying symbolically the instance and extracting a combinatorial core which it solves via the SAT solver minisat.

Table 2. Comparison between QBF and SAT based BMC on some industrial benchmarks. The encoding and benchmark set are as described in [55]. Time is measured in seconds. The "Mem." columns give the footprint in megabytes of the file containing the instance. TO stands for timeout (1000 seconds allotted). MO means that either the generator or the solver failed for an out-of-memory condition (2 GB allotted). The solvers used are sKizzo v-0.10 and minisat v-1.14.

Circuit 1 / Property 2 (all false)

	sKizzo on QBF		minisat on SAT	
k	*Time*	*Mem.*	*Time*	*Mem.*
2	0.6	1.0	0.2	1.3
4	1.3	1.4	0.4	2.6
8	21.4	2.4	1.0	5.5
16	40.5	4.6	1.5	11.6
32	29.9	5.3	3.3	23.8
64	22.2	6.9	4.9	48.7
128	43.9	10.9	10.27	103.3
256	113.4	18.2	22.49	212.6
512	236.2	32.9	—	MO
1024	947.3	128.9	—	MO

Circuit 2 / Property 1 (all true)

	sKizzo on QBF		minisat on SAT	
k	*Time*	*Mem.*	*Time*	*Mem.*
2	0.4	0.3	0.2	0.5
4	7.3	0.4	0.2	1.0
8	11.6	0.6	0.4	2.1
16	46.4	1.2	1.6	4.2
32	13.6	1.6	10.6	8.9
64	23.0	2.2	314.4	19
128	66.9	3.4	TO	38
256	44.0	5.9	TO	79
512	TO	12.0	TO	164
1024	TO	23.0	—	MO

Circuit 6 / Property 1 (all true)

	sKizzo on QBF		minisat on SAT	
k	*Time*	*Mem.*	*Time*	*Mem.*
2	0.1	0.1	0.1	0.1
4	0.2	0.1	0.1	0.3
8	0.3	0.2	0.2	0.7
16	1.0	0.3	0.4	1.4
32	1.3	0.4	1.2	2.8
64	2.1	0.6	6.7	5.9
128	3.6	1.1	25.1	13
256	10.7	1.9	238.3	26
512	57.1	3.6	TO	52
1024	864.3	7.3	TO	110

Circuit 6 / Property 2 (all false)

	sKizzo on QBF		minisat on SAT	
k	*Time*	*Mem.*	*Time*	*Mem.*
2	0.1	0.1	0.1	0.1
4	0.1	0.1	0.1	0.3
8	0.1	0.2	0.2	0.7
16	0.2	0.3	0.4	1.4
32	0.3	0.4	0.5	2.8
64	0.5	0.6	0.9	5.8
128	0.8	1.0	1.9	12.3
256	1.4	1.9	3.6	25.2
512	2.7	3.6	6.6	51.5
1024	5.2	7.2	13.35	109.3

A more interesting comparison is the one against SAT-based solutions for the same instances. Table 2 reports a few representative comparisons from the set of industrial benchmarks considered in Figure 9. There is some variation in the relative strength of the SAT-based and QBF-based formalization. However, *the results are not unfavorable to QBF*. In some cases (e.g., Circuit 6 / Property 2), the two approaches are almost equivalent, with the slight advantage of the QBF approach perhaps only due to the large input size of the equivalent SAT instance. In other cases (e.g., Circuit 1 / Property 2) there is a slight advantage for the SAT-based approach, but the QBF-based one returns to lead as soon as memory consumption becomes a bottleneck for SAT. On the most complex cases, we observe a behavior favorable to the QBF version (e.g., Circuit 2 / Property 1 and Circuit 6 / Property 1), even when memory is not the critical resource.

3.3 The Role of Certificates

What are the applications of QBF certificates to FV tasks? Resolving conflicting answers given by different solvers on the same FV problem is the first application that comes to mind[12.]. Yet, a certificate is much more than a way to ensure validity: It can be *inspected* to gather information about the model it represents. In this sense, QBF certificates are just like the widely employed "certificates" for SAT instances, i.e., satisfying assignments to their variables[13.]. Beyond replicating SAT-inspired practices, QBF models permit—by their tree-like nature—to represent strategies, a useful feature in most game-like scenarios: The rules of the game and the existence of a winning strategy can be encoded into a QBF (see, e.g., [39]), and the related certificate explicitly represents such a strategy. Game-like scenarios are not necessarily perceived as games by humans: *Conformant planning* [66], for example, and many FV problems (Section 2.2) are essentially games.

In particular, three roles that certificates may play in FV are (in increasing order of abstraction):

1. The certificate can be seen as a compact repository for skolem function interpretations, to be queried when we need the values of "deep" existentials as a function of preceding universals; depending on the encoding, such deep existential variables may bear information which is important to explicitly solve a FV task. For example, they may encode counterexamples which are otherwise (i.e., using a QBF solver which does not produce certificates) difficult to extract. An example of this scenario is discussed in Section 3.3.1;
2. Some portions of the information contained in the certificate, when properly combined, can provide a compact characterization of scenarios in which interesting conditions hold; for example, if the counterexample showing a property violation is embedded in deep existentials as discussed before, we may operate on the certificate in so as to extract compact answers for questions such as "in which time instants along the bad path this variable was in a don't-care condition?". Examples are discussed in Section 3.3.2;
3. The whole certificate of some properly shaped QBF—thought of as the output of an automated synthesis process—may be an interesting piece of information by itself. In

12. These soundness issues occur fairly often in practice, for no finer problem than bugs in the implementation. Certificates greatly help to track bugs: QBF solvers are indeed quite complex pieces of software, and the semantics trees of QBFs from applications are so big that an automatized certification approach is the only realistic way to tell the truth.
13. As opposed to QBF certificates, SAT certificates are easy to represent and verify, hence they have had a wide application (virtually every SAT solver is able to exhibit such certificates). For example, a sat answer to the propositional (PROP) encoding of (the negation of) a desired property over a logic circuit means that the circuit is faulty w.r.t. that property. But, it takes a certificate to outline a definite scenario in which the fault shows up. In general, the *bounded model checking* (BMC) technique for LTL properties uses the certificate produced by the SAT solver to construct a witness that violates a desired property. Furthermore, some abstraction/refinement modeling frameworks exploit certificates of satisfiability over (too) abstract versions of a system to refine its model.

this sense, a certificate is the automatically synthesized implementation of a circuit (based on BDD decision nodes, i.e., on if-then-else gates) which computes the function implicitly and declaratively characterized in the certified QBF. An example of this application is given in Section 3.3.2.

The first setting can be put to good use immediately, and is exemplified through a valuable real-world application. Proof-of-concept exemplifications are provided for the other two cases.

3.3.1 Certificates as Skolem Interpretations

The most direct usage of certificates consists of computing values for existential variables as a function of preceding universal ones, i.e., to exploit them as repositories of valid interpretations for the skolem terms. When and if this computation makes any sense, depends on the structure of the underlying encoding. It is not difficult, anyway, to sort out examples of usage in the FV domain. Let us reconsider, for example, the QBF-based BMC encoding built after the expression (22) in Section 3.2:

$$\begin{aligned}&\exists s_0,\ldots,s_{2^n}\ I(s_0)\wedge B(s_{2^n})\wedge\\ &\qquad\wedge\ \forall t\ \exists s,s'.\ [mux_n(s,t,s_0,\ldots,s_{2^n-1})\wedge mux_n(s',t,s_1,\ldots,s_{2^n})\wedge T_w(s,s')]\end{aligned}\tag{24}$$

where

$$T_w(s,s')\equiv\exists x_1,\ldots,x_{w-1}.\ T(s,x_1)\wedge T(x_1,x_2)\wedge\cdots\wedge T(x_{w-1},s')\tag{25}$$

Suppose we discover that an instance of (24) is true. This means that the underlying finite state machine can be driven from an initial state into a bad state, after 2^n steps. Which way, exactly?

To recover the full path—which is a key information to debug the system—we need to know the state of the machine at each time step. Such information is only partially specified by the standard feedback we obtain when QBF solvers prove that (24) is true. The feedback consists of a valid assignment for the outermost existential variables in (24), but these variables only describe a (possibly small) subset of the path (see Section 3.2). In the encoding we are using, the complete path is traced by the values that the variables x_i in (25) assume as a function of the universal variable t in the second scope of (24). So, to recover this information, we need to:

1. Solve the formula with a QBF solver able to produce valid assignments to the variables $s_0,\ldots,s_{2^n}$ in the outermost scope;
2. Force this assignment into the matrix of the formula;
3. Generate all the combinations of assignments to the universal variables in t (i.e., consider all the time frames separately), and for each assignment/frame we:
   (a) Force the assignment to t and solve the resulting formula with a SAT solver;
   (b) Extract from the model the truth values of the variables x_i in the current frame.

We are easily convinced that this method is not practical by observing that (i) it requires to call a QBF solver anyway (step 1), and then a SAT solver a number of times exponential in the size of t (step 3.a), thus essentially mimicking what a search-based QBF solver would do to decide the instance, (ii) the simulation of a search-based solver is necessarily (much) worse than the genuine solver (for it lacks many features of real, competitive solvers, such as sharing of information among frames by learned clauses, look-back enhancements such as backjumping or model caching, etc.), and (iii) even at their full potential, search-based solvers are already considerably weaker than alternative approaches for this problem (cfr. Figure 9, where search-based solvers are showed to be much slower than quantor and sKizzo on the encoding we are using here).

Fortunately, there is a much more direct and effective way to extract the path. If we solve the formula by a solver able to provide certificates, then what a certificate encodes is a valid interpretation for the functions in the skolemized version of (24). Such skolemized version is

$$I(s_0) \wedge B(s_{2^n}) \;\wedge\; \forall t.\ [mux_n(f(t), t, s_0, \ldots, s_{2^n-1}) \wedge mux_n(f'(t), t, s_1, \ldots, s_{2^n}) \wedge \\ T(f(t), f_1(t)) \wedge T(f_1(t), f_2(t)) \wedge \cdots \wedge T(f_{w-1}(t), f'(t)))]$$

where: the constants $s_0, \ldots, s_{2^n-1}$ have been assigned for simplicity the same symbol as the existential variables they relate to; the functions $f(t)$ and $f'(t)$ skolemize s and s' respectively (t is in general a vector of bits, rather than a single variable); the functions $f_1, \ldots, f_{w-1}$ are the skolem terms associated with the variables $x_1, \ldots, x_{w-1}$ in (25). While most QBF solvers are able to compute the values of the skolem constants $s_0, \ldots, s_{2^n-1}$, the values that the functions $f_1, \ldots, f_{w-1}$ assume at all points in their definition domains are necessary to describe the entire path. A certificate encodes all such values, so it outlines a definite path leading the machine into a bad state.

Is this certificate-based process viable? To retrieve relevant information from a certificate, e.g., to compute $f_i(t)$ for specific values of i and t, we only need to perform linear-time BDD operations. However, the construction and validation of the certificate itself may be critical, because QBF certification is intractable in general [27]. Fortunately, in many practical cases it turns out that certification is an easy computational task, compared to solving the instance. Some results for the encoding we are considering are presented in Table 3. The table presents run-times and other information for the process of reconstructing a certificate which encodes counterexamples of different lengths for a faulty circuit (the same case studies and properties as in Section 3.2 are considered). The results show that the activities related to certification (construction and validation of the certificate, requiring total run-time T_r+T_v) have a minimal impact on the overall run-time (T_s+T_r+T_v). All the details of the bad paths are thus extracted with a negligible overhead.

3.3.2 Advanced Usage of Certificates

Certificates can be used in more sophisticated ways than previously shown. A first approach is to think of certification as an *automated synthesis* step: The certificate is considered as a "computational device" to be synthesized. From this perspective, the QBF solver/certifier is exploited as an engine that, given a declarative specification of some defining features for a function of interest, performs the synthesis of a circuit (based on if-then-else gates) which implements that function. For example, the certificates of Pan's benchmarks, according to the description of this family given on Page 141, are implementations of barrel-shifters

Table 3. Certificate reconstruction and verification for families of true BMC instances from Section 2.3. We report: the number of variables and clauses, the shape of the prefix, the time taken to solve/reconstruct/verify a certificate ($T_s/T_r/T_v$), the size of the log ($|\mathcal{L}|$, number of inference steps written in the log by the solver) and of the certificate ($|\mathcal{C}|$, overall number of decision nodes in the forest of BDDs which represents the certificate).

instance	vars	clauses	prefix	T_s	T_r	T_v	$\|\mathcal{L}\|$	$\|\mathcal{C}\|$
c6_BMC_p1_k2	2282	6263	∃∀∃	0.1	0.1	0.1	2405	2
c6_BMC_p1_k4	2463	6983	∃∀∃	0.2	0.1	0.1	2910	8
c6_BMC_p1_k8	2824	8423	∃∀∃	0.3	0.1	0.1	3490	20
c6_BMC_p1_k16	3545	11303	∃∀∃	1.0	0.2	0.1	5177	92
c6_BMC_p1_k32	5555	16664	∃∀∃	1.3	0.2	0.1	7288	66
c6_BMC_p1_k64	9753	27746	∃∀∃	2.1	0.4	0.2	11850	96
c6_BMC_p1_k128	18149	49910	∃∀∃	3.6	0.6	0.3	20675	89
c6_BMC_p1_k256	34941	94238	∃∀∃	10.7	1.0	0.6	38390	88
c6_BMC_p1_k512	68525	182894	∃∀∃	57.1	1.8	1.0	73822	249
c6_BMC_p1_k1024	135693	360206	∃∀∃	864.3	3.5	2.1	144656	914

instance	vars	clauses	prefix	T_s	T_r	T_v	$\|\mathcal{L}\|$	$\|\mathcal{C}\|$
c2_BMC_p1_k2	6414	18882	∃∀∃	0.1	0.1	0.1	7186	2
c2_BMC_p1_k4	7191	21986	∃∀∃	7.3	0.4	0.1	14924	8
c2_BMC_p1_k8	8744	28194	∃∀∃	11.6	0.6	0.2	18126	50
c2_BMC_p1_k16	11849	40610	∃∀∃	46.4	1.1	0.4	25761	305
c2_BMC_p1_k32	17097	55610	∃∀∃	13.6	1.2	0.5	29892	511
c2_BMC_p1_k64	28367	87162	∃∀∃	23.0	1.7	0.7	47331	267
c2_BMC_p1_k128	50907	150266	∃∀∃	66.9	3.0	0.9	81586	231
c2_BMC_p1_k256	95987	276474	∃∀∃	44.0	5.0	1.6	149590	204

(see also the related application presented in [54]). Let us develop a detailed example of certificate-based synthesis.

Example 8 *Let us encode in QBF the following statement:*

$$\forall a \in [0,\ldots,2^n)\ \forall b \in [0,\ldots,2^n)\ \exists c \in [0,\ldots,2^n).\quad c = (a+b)_{mod(2^n)} \tag{26}$$

where a, b and c are bounded integers. This true formula declares the existence of a result in $[0,\ldots,2^n)$ for the modular sum $(a+b)_{mod(2^n)}$, so it defines the integer sum modulo 2^n of two bounded n-bit integers. A Boolean encoding of (26) can be obtained by using, for any given n, the binary representation $a = \langle a_{n-1}\ldots a_1a_0\rangle$, $b = \langle b_{n-1}\ldots b_1b_0\rangle$, and $c = \langle c_{n-1}\ldots c_1c_0\rangle$, in so as to have $a = \sum_{i=0}^{n-1} a_i \cdot 2^i$, $b = \sum_{i=0}^{n-1} b_i \cdot 2^i$, and $c = \sum_{i=0}^{n-1} c_i \cdot 2^i$. A set of Boolean constraints which captures the sum of two bounded integers in terms of a_i, b_i, c_i can be written in several ways. If we introduce the auxiliary carry *variables $r = \langle r_{n-1}\ldots r_1r_0\rangle$, one classical version requires that:*

1. *the first carry is false: $\neg r_0$;*

2. the $(i+1)^{th}$ bit in the carry is true when at least two out of the three bits a_i, b_i, r_i are true: $\bigwedge_{i=0,\ldots,n-1} r_{i+1} \leftrightarrow ((a_i \wedge b_i) \vee (a_i \wedge r_i) \vee (b_i \wedge r_i)))$

3. the invariant linking the i^{th} bit in a, b, c and r is the existence of an even number of true values: $\bigwedge_{i=0,\ldots,n-1} \neg xor(a_i, b_i, c_i, r_i)$

These constraints—conjoined and put in the scope of the quantification $\forall a_{n-1}, \ldots, a_0\ \forall b_{n-1}, \ldots, b_0\ \exists c_{n-1}, \ldots, c_0\ \exists r_{n-1}, \ldots, r_0$—constitute a true QBF which is the Boolean translation of (26). We recognize that a certificate of satisfiability for such QBF shows by construction how to compute c as a function of a and b, hence it is the implementation of an adder. For $n = 8$ we obtain the model/certificate/implementation in Figure 10. The model reconstructor has been requested to minimize the number of gates/nodes in the model, and the portion of the certificate concerned with carries has been omitted for simplicity. Note how the best bit interleaving and circuit structure for the implementation of the adder has been automatically extracted from a declarative specification.

Real-world applications may include the certificate-based synthesis mechanism as a basic building block of some larger FV task. One such method has been recently introduced [79]. The details of the approach are quite involved, so we refer the reader to [79] for a through presentation. Here we limit ourself to a high-level description of the technique. The FV task considered in [79] bears some resemblance with the one we will examine in Section 3.4.3, though the formalization and the solution identified are completely different. In particular, the problem is to analyze a sequential circuit, determine whether some component is faulty (i.e., entirely responsible for some bad behavior of the system), and, in case it is, synthesize a repair—if one exists—which can be implemented by a substitute (deterministic) component. A component may be a single gate or some larger portion of the circuit. To ensure that the repairing component is indeed a repair *for all* the possible input sequences to the circuit, a QBF formalization is necessary. Then, a replacement for the faulty component is automatically synthesized by extracting its structure from a QBF certificate.

Yet another possibility to extract information from certificates is to define the *evaluation of expressions over certificates*. Let us call *scenario* any set of assignments to the universal variables of a formula F, and *expression* any Boolean expression over the existential variables of F. Given a certificate C and a scenario U for F, we perform a *direct expression evaluation* of the expression E by extracting from C the skolem functions associated with the variables that appear in E, computing their truth values under the assignment(s) in U, and combining these truth values according to E, to obtain the result of the evaluation (a truth value). Conversely, an *inverse expression evaluation* associates to a given expression E the largest scenario U in which it is true w.r.t. C.

Inverse expressions evaluate to sets of assignments to the universal variables, which are naturally represented by employing BDDs with universal variables as decision variables, coherently with the representation used by certificates. Direct expressions, when evaluated over a single total assignment to the universal variables, yield one of the truth values $\{\texttt{T}, \texttt{F}, \texttt{DC}\}$, just like the skolem interpretations they are built upon (see Section 2.4.2). The result of the evaluation over scenarios which contain partial assignments is more complex to define, and is described with an example.

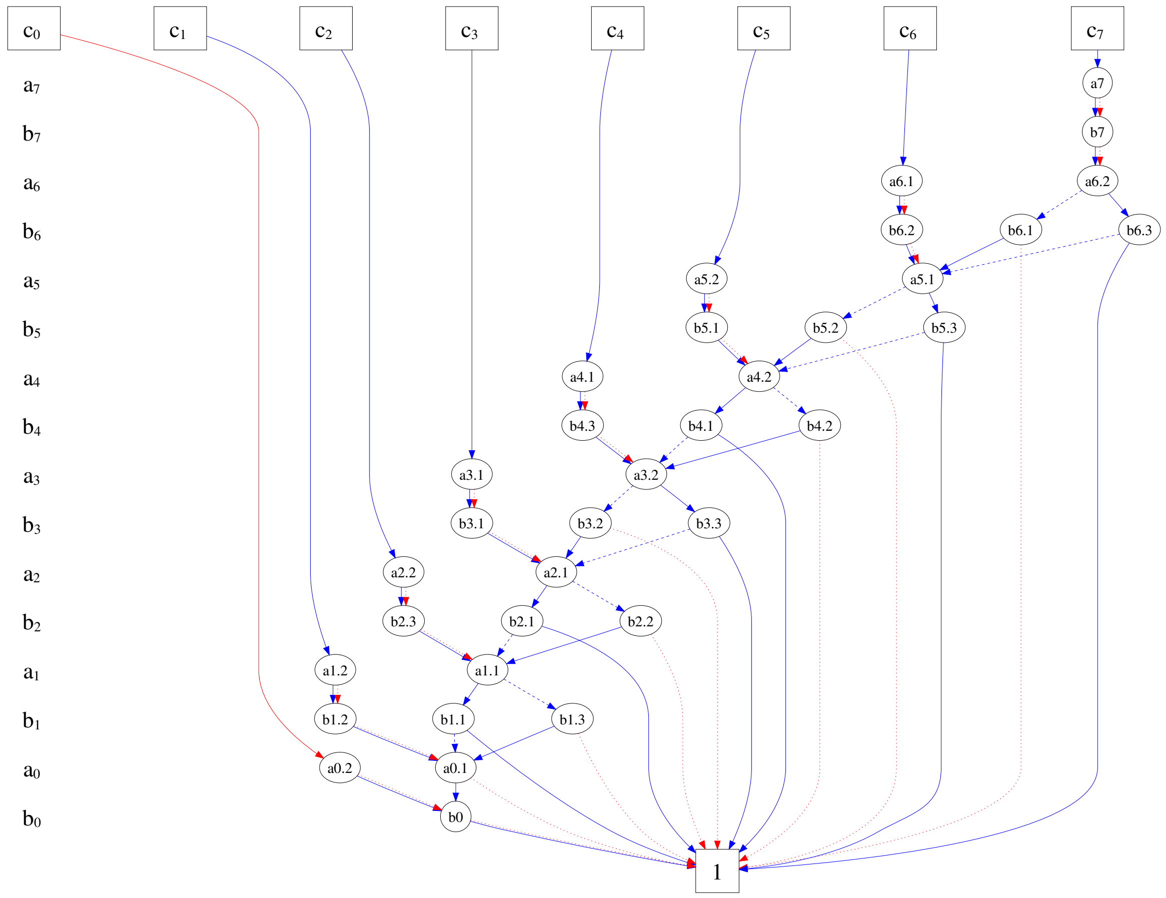


Figure 10. An 8-bit adder based on if-then-else gates synthesized as a side effect of QBF certification. The inputs of the circuit, listed in the column on the left, are the bits that encode the two addends $a = \langle a_7 \dots a_1 a_0 \rangle$ and $b = \langle b_7 \dots b_1 b_0 \rangle$. For each $i = 0, \dots, 7$, the internal nodes lying at the same vertical level as a_i and b_i are decision nodes on the variables a_i and b_i respectively. The arrangement for the inputs (bits are interleaved from the most to the least significant ones) is automatically generated to determine the smallest possible circuit. The outputs of the circuit are the bits $c = \langle c_7 \dots c_1 c_0 \rangle$, associated with the root notes in the topmost line, and are computed from the inputs according to the usual ROBBDs rule (described on page 148). The output bits encode the result of $(a+b)_{mod(256)}$, because the certificate ensures that $\forall a \forall b \exists c.\ c = (a+b)_{mod(256)}$ is true by showing how to compute c as a function of a, b in so as to comply with the constraint $c = (a+b)_{mod(256)}$.

Example 9 *Figure 11 presents examples of direct and inverse expression evaluations over the certificate for the QBF (14) given in Figure 2.*

Let us show, for example, how the expression $\neg e(a,b,c) \vee (\neg c(a,b) \wedge f(a,b,c))$ in the last line evaluates to $\boldsymbol{F}$ under the total universal assignment $a = 0, b = 1, d = 0$. The values of the valid skolem interpretations for the existential variables e, c, and f in the certificate

expression	*scenario*	*result*
$e(a,b,d)$	$a, \neg b$	DC
$e(a,b,d)$	$\neg a, \neg d$	F
$e(a,b,d)$	$a, b, \neg d$	T
$(c(a,b) \oplus e(a,b,d))$	$\neg d$	F
$(c(a,b) \oplus e(a,b,d))$	$a, \neg b$	wF
$(c(a,b) \oplus e(a,b,d))$	$a, \neg b, d$	DC
$(\neg e(a,b,d) \vee (\neg c(a,b) \wedge f(a,b,d))$	d	wT
$(\neg e(a,b,d) \vee (\neg c(a,b) \wedge f(a,b,d))$	$a, \neg d$	T
$(\neg e(a,b,d) \vee (\neg c(a,b) \wedge f(a,b,d))$	$\neg a, b, \neg d$	F

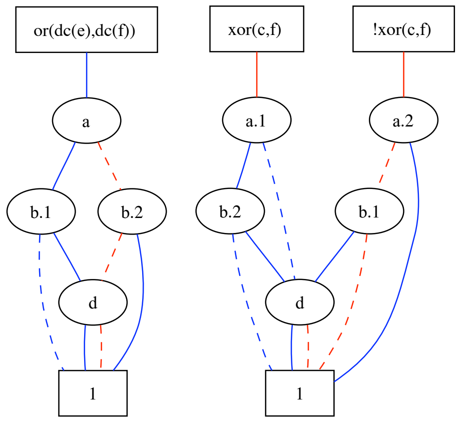


Figure 11. Examples of expression evaluations over the certificate in Figure 2 for the QBF (14). Example (9) explains how the results have been computed. *On the left*: three direct expressions evaluated over total and partial scenarios; *On the right*: evaluation of two inverse expressions over the same certificate. The first graph represents scenarios in which either e or f are in a don't-care condition. The second one recognizes whether the exclusive or between c and f is true/false.

in Figure 2, computed in the point $a = 0, b = 1, c = 0$ of their definition domain, namely $e = s_e(0,1,0) =$ T, $c = s_c(0,1) =$ T, and $f = s_f(0,1,0) =$ F, are such that the expression $\neg e \vee (\neg c \wedge f)$ is false.

*For partially specified scenarios, the result of the evaluation is more complex, because we have to consider all the total universal assignments that can be obtained by extending the given partial one. Let us consider, for example, the expression $c(a,b) \oplus e(a,b,d)$ under the partially specified scenario $a = 1, b = 0$ (fifth line in Figure 11). While $s_c(1,0)$ univocally evaluates to F, the value of $s_e(1,0,d)$ depends in general on the value assigned to d. We thus obtain several different results for the evaluation of the expression, one for each possible way to complete the partial universal assignment $a = 1, b = 0$. If two results that contradict each other exist (for example, $s_c(1,0) \oplus s_e(1,0,0)$ is true and $s_c(1,0) \oplus s_e(1,0,1)$ is false), we say that the partial expression $c(a,b) \oplus e(a,b,d)$ is not defined for $a = 1, b = 0$. If all the results comply (for example, both $s_c(1,0) \oplus s_e(1,0,0)$ and $s_c(1,0) \oplus s_e(1,0,1)$ are true), then the overall expression evaluates to the common value. Other two possibilities exist: The value wT (*weakly true*) is used to mean that in all the total universal assignments that extends the given partial one, the expression is either true, or in a don't-care condition. Similarly, wF stands for* weakly false*, and means that the expression is either false, or in a don't-care condition. In our sample case, it is $s_c(1,0) \oplus s_e(1,0,0) =$ F and $s_c(1,0) \oplus s_e(1,0,1) =$ DC, so the result is wF.*

The software ozziKs [10] has been extended to compute the result of direct and inverse expressions over total and partial scenarios, and to present results in different formats.

Can expressions over certificates be useful in FV? Though no application has been developed yet, examples of usage are easy to envision. For example, the inverse expression $\neg c_7 \vee \wedge_{i=0}^{6} \neg c_i$, evaluated over the certificate in Figure 10, computes all the couples of positive integers which add up to no more than 128; a few direct expressions evaluated over the formalization presented in Section 3.1, equations (18 − 19), define precisely which smaller strategic set disproves the importance of a given company; in BMC formalizations like the ones in Section 2.2 or Section 3.2, inverse expressions allow the characterization of all the frames or time points along a bad path in which a given Boolean condition over the finite state machine is true.

The key point that affects the feasibility of all these certificate-based applications is, again, the resource consumption of certification on families of true instances from practical FV tasks. Some results are reported in Table 4. Though there exist rare families—notably the sample $s499$ series from the Mneimneh and Sakallah's benchmarks—for which the overhead due to the compilation of interpretations is quite large[14.], certification appears to be feasible in general.

3.4 Open QBFs

All the QBFs we have considered so far are *closed* formulas, i.e., formulas in which every variable that appears in the matrix is existentially or universally quantified somewhere in the prefix. In this section we focus on QBFs containing *free variables*, i.e., variables in the scope of no quantifier. Such formulas are called *open QBFs*. We characterize the role of free variables and their relationship to *all-solution* solvers (Section 3.4.1), briefly consider how QBF solvers can be accommodated to open formulas (Section 3.4.2), and present an example of a real-world FV application in which open QBFs are used (Section 3.4.3).

3.4.1 Free Variables and All-Solution Solvers

Let us consider a satisfiable SAT instance F with variables V. Depending on the problem encoded in F, we may be interested in extracting via a SAT solver: (1) a simple yes/no answer to the question of whether the instance is satisfiable, (2a) a complete satisfying assignment to V, (2b) a *valid* assignment to some interesting subset $W \subset V$, i.e., one assignment to W which can be extended to at least one complete satisfying assignment to V, (3a) the set of all the complete satisfying assignments, or (3b) the set of all the distinct valid assignments to an interesting subset $W \subset V$[15.].

A QBF solver is also, as a special case, a SAT solver. So, an all-solution QBF solver should have the ability to confront tasks 1, 2a, 2b, 3a, and 3b as special cases. How can these different tasks be characterized, from a logical viewpoint, in a way that extends unaltered to QBF?

We assume the following perspective. The meaning of a formula is the "concept" it captures, i.e., the set of its models, i.e., the set of *interpretations* under which the formula is

14. This is not much of an issue for the $s499$ series itself, because according to (10) the skolem terms are encoding all the ways in which paths of length $k-1$ fail to reach states at depth k, an information with probably no practical utility.

15. Tasks (3a) and (3b) are carried out by the so called *all-solution* SAT solvers. The simplest way to build an all-solution solver is to call a standard SAT solver several times, adding *blocking clauses* [58] in between runs to prevent solutions from appearing more than once. All-solution SAT solvers have been applied to many FV tasks, including test pattern generation [50], circuit minimization [73], model checking [58], and design debugging [4].

Table 4. Certification of families of FV instances from Section 2.3. We report: the number of variables and clauses, the shape of the prefix, the time taken to solve/reconstruct/verify a certificate ($T_s/T_r/T_v$), the size of the log ($|\mathcal{L}|$, number of steps) and of the certificate ($|\mathcal{C}|$, number of nodes). No timeout limit was imposed.

instance	vars	clauses	prefix	T_s	T_r	T_v	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{C}\rvert$
adder-2	332	113	∀∃∀∃	0.1	0.1	0.1	22	$1.0 \cdot 10^3$
adder-4	726	534	∀∃∀∃	0.2	0.1	0.1	165	$1.9 \cdot 10^3$
adder-6	1272	1263	∀∃∀∃	1.1	2.7	0.1	384	$7.6 \cdot 10^3$
adder-8	1970	2300	∀∃∀∃	5.4	19.4	0.1	696	$1.6 \cdot 10^4$
adder-10	2820	3645	∀∃∀∃	30.0	86.0	0.1	1099	$3.6 \cdot 10^4$
adder-12	3822	5298	∀∃∀∃	130.2	391.5	0.2	1688	$8.1 \cdot 10^4$
adder-14	4976	7259	∀∃∀∃	260.0	1091.3	0.6	2277	$1.7 \cdot 10^5$
adder-16	6282	9528	∀∃∀∃	710.0	4026.2	1.1	3761	$3.2 \cdot 10^5$

instance	vars	clauses	prefix	T_s	T_r	T_v	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{C}\rvert$
cnt06	266	691	$\exists(\forall\exists)^6$	0.1	0.1	0.1	16	42
cnt07	352	918	$\exists(\forall\exists)^7$	0.2	0.1	0.1	17	56
cnt08	450	1177	$\exists(\forall\exists)^8$	1.0	0.7	0.1	21077	72
cnt09	560	1468	$\exists(\forall\exists)^9$	3.5	1.6	0.1	47311	90
cnt10	682	1791	$\exists(\forall\exists)^{10}$	8.4	2.7	0.1	74736	131
cnt11	816	2146	$\exists(\forall\exists)^{11}$	18.0	4.2	0.1	111101	132
cnt12	962	2533	$\exists(\forall\exists)^{12}$	31.0	8.2	0.1	204370	156
cnt13	1120	2952	$\exists(\forall\exists)^{13}$	36.0	8.7	0.1	200428	184
cnt14	1290	3403	$\exists(\forall\exists)^{14}$	39.0	12.5	0.1	280926	210
cnt15	1472	3886	$\exists(\forall\exists)^{15}$	41.0	14.8	0.1	232420	329
cnt16	1666	4401	$\exists(\forall\exists)^{16}$	84.0	35.8	0.1	679529	385

instance	vars	clauses	prefix	T_s	T_r	T_v	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{C}\rvert$
s499_s2_s	1213	2665	∃∀∃	0.1	2.1	0.1	987	505
s499_s3_s	2545	4816	∃∀∃	0.4	31.5	0.1	1860	1324
s499_s4_s	4868	6967	∃∀∃	1.1	66.7	1.1	3371	2140
s499_s7_s	12783	13420	∃∀∃	5.6	459.8	5.8	6705	4600
s499_s10_s	25617	19873	∃∀∃	15.0	2112.3	26.0	9990	7046
s499_s14_s	49603	28477	∃∀∃	38.2	1109.2	76.4	14285	10328
s499_s18_s	81445	37081	∃∀∃	84.1	11081.7	177.7	18485	13608
s499_s19_s	90633	39232	∃∀∃	94.6	12186.9	205.9	19523	14428

instance	vars	clauses	prefix	T_s	T_r	T_v	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{C}\rvert$
qshifter_3	19	128	∀∃	0.1	0.1	0.1	10	33
qshifter_4	36	512	∀∃	0.1	0.1	0.1	18	81
qshifter_5	69	2048	∀∃	0.1	0.1	0.1	34	193
qshifter_6	134	8192	∀∃	0.2	0.1	0.1	66	449
qshifter_7	263	32768	∀∃	1.0	0.4	0.4	130	1025
qshifter_8	520	131072	∀∃	6.2	2.4	1.7	258	2305

true. Interpretations are Boolean assignments to the non-quantified variables in the formula. All the other variables—quantified either existentially or universally—are considered as auxiliary logic tools which are instrumental in expressing the meaning of the formula, but do not contribute directly to the structure of models. In this sense, quantifiers entirely absorb the meaning of the variables they bind.

This perspective justifies the following characterization, shared by SAT and QBF:

decision problem: the yes/no *validity problem* (case 1) is the problem of deciding whether the meaning of a *closed* formula is empty;

search problem: the *one-solution problem* (cases 2a and 2b) is the problem of finding a valid assignment for the *free* variables of an *open* formula, i.e., to exhibit a model;

enumeration problem: the *all-solution problem* (cases 3a and 3b) is the problem of expressing the entire meaning of an *open* formula (w.r.t. its free variables).

Accordingly, a QBF solver fed with (possibly open) QBFs is able to answer the validity, one-solution, and all-solution problems, uniformly for SAT and QBF, as follows:

SAT A yes/no answer on the satisfiability of the formula (case 1) is what we obtain from a closed formula where all the variables are existentially quantified. A complete satisfying assignment (case 2a) is obtained by leaving all the variables free and solving the one-solution problem. The set of all the complete satisfying assignments (case 3a) is obtained by leaving all the variables free and solving the all-solution problem. One or all the distinct valid assignments to an interesting subset $W \subset V$ of the variables (cases 2b and 3b) are obtained by leaving the variables W free and quantifying existentially the uninteresting variables ($V \setminus W$).

QBF A yes/no answer on the validity of the formula (case 1) is what we obtain from a closed formula where all the variables are existentially or universally quantified. This is the default behavior of most QBF solvers, which only accept closed formulas as their input. When we solve the one-solution problem for an open formula (cases 2a and 2b), we obtain one valid assignment to its free variables, i.e., an assignment which, once applied to the matrix, makes the resulting closed QBF true. Some QBF solvers return a valid assignment to the variables in the outermost existential scope of true formulas. This behavior is in fact better understood if we consider the variables for which an assignment is returned as *open* (and the remaining ones as quantified). A formula may thus have both free variables and existential variables in the outermost scope (case 2b), and in this case a valid assignment to the free variables only is returned. In the most general case, we solve the all-solution problem for an open QBF (case 3a and 3b), thus obtaining all the distinct valid assignments to its free (or "interesting") variables.

Example 10 *Let us consider a quantifier-free formula* $M(a, b, c, d)$ *with models* $\{\{\neg a, b, \neg c, d\}, \{a, \neg b, c, d\}, \{a, \neg b, \neg c, \neg d\}\}$. *A solver for open QBFs is expected to compute the values in the column named "Result", given as input the QBF in the column "Formula":*

Problem	Formula	Type	Result	Comment
Validity check	$\exists a \exists b \exists c \exists d\ M(a,b,c,d)$	*closed*	*yes*	*standard SAT solver*
Validity check	$\exists a \forall b \forall c \exists d\ M(a,b,c,d)$	*closed*	*no*	*standard QBF solver*
One-solution	$\exists c \exists d\ M(a,b,c,d)$	*open*	$\{a=0, b=1\}$	*case 2b for SAT*
All-solutions	$\exists a \exists b \exists c\ M(a,b,c,d)$	*open*	$\{\{d=0\},\{d=1\}\}$	*case 3b for SAT*
One-solution	$\forall c \exists d\ M(a,b,c,d)$	*open*	$\{a=1, b=0\}$	*case 2b for QBF*
All-solutions	$\exists a \forall d \exists c\ M(a,b,c,d)$	*open*	$\{\{b=0\}\}$	*case 3b for QBF*

Formally, we pose the following definition. Given a CNF F with variables $V = var(F)$, we denote by $2^{lit(V)}$ the set of all the possible (partial) truth assignments to V (represented as sets of consistent literals), and by $A{*}F$ the formula resulting after the assignment $A \in 2^{lit(V)}$ is applied to F.

Definition 3.1 (Valid assignments and meaning of an open QBF) *Let M be a CNF and $F = \mathcal{Q}_1 V_1 \cdots \mathcal{Q}_n V_n.\ M$ be a prenex QBF with a set of* free variables $V_{free} = var(M) \setminus \cup_{i=1}^{n} V_i$*. We say that an assignment $A \in 2^{lit(V_{free})}$ to the free variables V_{free} of F is* valid *for F if $\mathcal{Q}_1 V_1 \cdots \mathcal{Q}_n V_n.\ (A * M)$ is a true QBF. The meaning of F is the set of all the valid assignment to V_{free}.*

In the next section we describe how we extended sKizzo to comply with such characterization: The extended solver is able to work in *decision* mode (to decide whether a formula has any model at all), in *search* mode (to find a model of a SAT/QBF instance), and in *enumeration* mode (to find all the models of a SAT/QBF instance).

3.4.2 Solvers for Open QBFs

There is no major complication in adapting existing QBF solver architectures to open QBFs. However, the adjustments largely depend on the inference strategy adopted by the solver. Hereafter we informally describe how we modified the inference engines of sKizzo. These extensions cover all the categories of QBF reasoners we are aware of, and can be easily replicated within other solvers.

DPLL-like search. In the spirit of [44], free variables are treated as if they were in a "special" scope, to the left of (i.e., dominating) all the variables in the prefix. This means that the search procedure assigns a (tentative) truth value to all the free variables before any branching over quantified variables takes place. A depth threshold thus exists in the search stack beyond which all the free variables have got a truth value, and the QBF becomes closed. By pushing splits over free variables towards the shallowest recursion levels in the evaluation tree, we are ensured that a valid assignment to all these variables is discovered every time the procedure backtracks below the above mentioned threshold bearing a positive evaluation. This also allows to share meaningful learnt clauses among subsequent searches for valid assignments.

To ensure that *all* the valid assignments are discovered, a complete search over the Boolean space induced by the free variables is required:

- Both branches of search nodes associated with free variables are to be visited (like universal nodes, and unlike existential nodes);

- A negative answer from the branch visited first does not prevent the other branch from being explored (like existential nodes, and unlike universal nodes).

Resolution. Quantified resolution in solvers such as quantor and sKizzo is employed within variable elimination rules. This means that resolution steps are arranged in atomic sequences. Each sequence consists of applying q-resolution [25] in all the possible combinations, among clauses containing a given (existential) literal on the variable e and clauses containing the opposite literal. The parent clauses are then discarded, and their resolvent children are kept. As an effect, variable e disappears, but validity is preserved. Free variables, in the scope of no quantifier, are treated as variables whose elimination is *prohibited*. The meaning of the open QBF is fully characterized when all the variables but V_{free} disappear. It suffices to either output the residual formula [25], or engage an all-solution SAT solver to list its models[16].

SAT-based reasoning. One or more *propositional expansions* [12] may be generated by sKizzo while solving QBF instances. The SAT compilation engine is instructed to signal which variables of a SAT compilation—if any—are related to free variables of the original QBF. Free variables, by definition, are not skolemized and are in the scope of no universal quantifier. So, their truth value depends on no universal variable, and this implies that free variables in the QBF map to free variables in the propositional expansion. All the distinct valid assignments to the free variables of this expansion are enumerated using an all-solution SAT solver.

Forward inferences. Forced assignments to free variables (i.e., a truth value for a variable in V_{free} whose disregard entails an inconsistency) can be discovered by inference rules such as unit clause propagation. These facts are recorded and included in all the valid assignments. The application to free variables of rules that do not preserve logical equivalence, such as pure literal elimination, is disabled in general, because we aim to find *all* the models. Some rules that do not preserve logical equivalence are applied anyway—such as equivalence-based substitutions[17]—provided a sufficient amount of information is recorded to restore a full model. In the case of equivalence-based substitutions, for example, we need to recover the truth value of all the free variables that have disappeared by substitution.

Hybrid solvers. The adaptation of each inference technique in isolation from the others is not hard. Some complications arise when a hybrid solver is to be extended to open QBFs. Inside sKizzo, for example, inference attacks are not tried sequentially. Rather, they are nested one inside the other [15]. As a result, portions of a single valid assignment to the free variables may be discovered by different inference engines, and may exhibit cross-dependencies.

16. Quantifier elimination is the most direct way to compute the meaning of an open QBF. Its main potential drawback is a common one among resolution-based methods: Despite a carefully designed elimination schedule, the amount of memory required to maintain intermediate representations during the elimination of $V \setminus V_{free}$ may become prohibitive.
17. A special case arises when a free variable is discovered to be equivalent to an existential variable. In general, there is no preferential way to apply substitution, but in this case the free variable replaces the quantified one.

To confront this complexity, we entrust a new module—called *Free Variable Manager* (FVM) and external to each inference engine—with three mutually complementary tasks:

Listen. This task consists of keeping track of:

- assignments to free variables by equivalence-preserving rules (SUCP, SBR);
- equivalence substitutions (SER) involving at least one free variable;
- search state of the DPLL engine, and dependencies of inferences on splits;
- migration of free variables into propositional expansions;
- possible decomposition of the problem into unrelated subproblems;
- order in which and reason why inference engines have questioned each other.

Coordinate. In this role, the FVM is meant to:

- prevent rules that compromise logical equivalence from operating on free variables;
- instruct the DPLL engine to branch on free variables first, and to consider both search branches in any case;
- suggest the orientation of equivalence-based substitution involving free variables;
- ban the elimination of free variables by resolution;

Recapitulate. This task consists of:

- recognizing when a complete valid assignment has been encountered, and
- combining all the information gathered during the listening task to assemble such a model. This step involves the use of a limited form of the *inductive model reconstruction* techniques described in [13].

The resulting solver is available for download [16], and is the one we use in the next section to experiment with an application of open QBFs to a formal verification task.

3.4.3 A Real-World Application for Open QBFs

An application of open QBFs we recently contributed to explore comes from *design debugging automation* (DD). Given a sequential circuit which produces erroneous answers, DD aims to locate (automatically) where the circuit's design needs a fix. SAT-based approaches to this task exist [4, 3, 77]. Here we briefly introduce and test their QBF formulation, fully described in [56].

Let us consider transition relations that explicitly mention inputs and outputs. In this framework, a transition relation $T(s, x, s', y)$ is used to represent the fact that a circuit in the state s (made up of h bits) to which inputs x are applied emits output y and switches to state s'. State and inputs, together, deterministically settle the next state and outputs, so that only one assignment to s' and y is consistent with any given assignment to s and x.

Suppose we discover some bad behavior of the resulting system after $k = 2^n$ steps (for example, by BMC). This means that the circuit, starting from state s_0 and evolving according to the inputs[18.] $\overline{x_1}, \ldots, \overline{x_k}$, produces a sequence of outputs $\overline{y_1^{err}}, \ldots, \overline{y_k^{err}}$ that is wrong as it differs in at least one point from the expected output sequence $\overline{y_1}, \ldots, \overline{y_k}$.

We can use the multiplexer-based way of marching along paths from Section 3.2, to enforce (a) the validity of the path by a single copy of T, and (b) the desired inputs and outputs of the system[19.]:

$$\exists s_0, \ldots, s_k \; \forall t \; \exists s, x, s', y. \; [mux_n(s, t, s_0, \ldots, s_{k-1}) \wedge mux_n(s', t, s_1, \ldots, s_k) \wedge \\ mux_n(x, t, \overline{x_1}, \ldots, \overline{x_k}) \wedge mux_n(y, t, \overline{y_1}, \ldots, \overline{y_k}) \wedge T(s, x, s', y)] \tag{27}$$

This formula asks whether there exist valid state transitions from s_0 to s_k, such that the (erroneous) system with transition relation $T(s, x, s', y)$ subject to inputs $\overline{x_1}, \ldots, \overline{x_k}$ produces as output the (correct) sequence $\overline{y_1}, \ldots, \overline{y_k}$. By the definition of the behavior of the erroneous transition relation T, (27) is false because T will instead produce the different sequence of outputs $\overline{y_1^{err}}, \ldots, \overline{y_k^{err}}$, and there is only one deterministic output for each sequence of inputs and states.

By replacing in (27) the transition relation T of the erroneous circuit with the representation T' of a correct circuit, we obtain a true formula since T' would then produce the expected sequence of outputs. How can we pinpoint the "smallest" possible modification T' to T which, substituted in (27), makes the formula evaluate to true? In other words: Which is the simplest fix for the system?

Let us start by introducing an *error selector* vector $e = \langle e_1, \ldots, e_h \rangle$ and a *rectifying* vector $w = \langle w_1, \ldots, w_h \rangle$, both consisting of as many bits as there are state bits in the system, namely h bits. The intuition is that the error selector vector will be used to identify "erroneous" state bits, and the rectifying vector will be used to provide a rectified value for such bits. We also define an *enhanced* transition relation T_{en} which behaves just like T, with the exception that if the bit e_i is true, then the i^{th} bit of the next state of the system as computed by T_{en} has to take the (arbitrary) value of the i^{th} bit w_i of the rectifying vector w, rather than the nominal value s'_i. The flexibility of adjusting any number of state bits by means of T_{en} is formally expressed as follows:

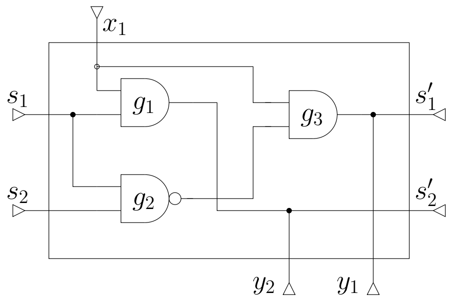
$$T_{en}(s, x, e, w, s', y) \equiv \exists r. [\wedge_{i=1}^{h} (e_i \rightarrow s'_i = w_i \;\wedge\; \neg e_i \rightarrow s'_i = r_i)] \;\wedge\; T(s, x, r, y)$$

Note that if all the h bits of the error selector vector e are false, then w has no impact on the enhanced transition relation T_{en}, which behaves as a faithful copy of T. Conversely, we inject more and more degrees of freedom into T_{en} by assigning some of the bits in e to true. By rectifying sufficiently many state bits, T_{en} will be made "different enough" from T to be able to exhibit the expected input/output behavior. These degrees of freedom can be linked one-to-one to locations in the design (hence to possible fault locations), by properly encoding the system. For example, in digital systems we can model the output of each gate by one state bit, and build the enhanced transition relation in such a way that each bit of the error selector vector is associated with one gate.

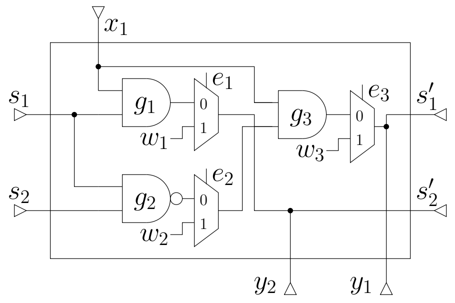
18. We put dashes over symbols which stand for parameters with a fixed value, rather than for variable names.

19. To simplify the exposition, we don't use the frame-based technique from Section 3.2.

Example 11 *Let us consider the following two circuits.*



(a) Erroneous circuit



(b) Enhanced circuit

These circuits, with one input (x_1) and two outputs (y_1, y_2), are the combinatorial sections of two sequential circuits, i.e., their next states s'_1, s'_2 are meant to be retained by two flip flops (not represented), from which they are fed back as s_1, s_2 at the next step. The idea is that circuit (a) exhibits the original transition relation T, while the enhanced transition relation T_{en} is associated with circuit (b) in a way that links the error selection/correction bits one-to-one to gates.

Suppose we know that circuit (a) is erroneous because, starting from the all-false initial state ($s_1 = 0$, $s_2 = 0$), the 2-step sequence of inputs 1, 1 for x_1 produces the output sequence $\langle 1, 0\rangle, \langle 1, 1\rangle$, while we know the correct circuit outputs $\langle 1, 0\rangle, \langle 0, 1\rangle$. What happened is that there is a bug in the design: Gate g_2 is a `NAND`*, but in fact it should be a* `NOR`*. It is easy to check that the correct output is produced by replacing g_2 with a* `NOR`*. In real settings we know that something is wrong because the output sequence does not match our expectations, but we have no idea how may bad gates are present in the circuit, which are their locations, and how to fix them.*

To solve this problem, we construct circuit (b), which is a multiplexer-based implementation of T_{en}, in which each true bit of the error selection vector e disconnects a gate and replaces its output by a rectified value from the corresponding bit in the rectifying vector w. By setting to true the error selector bits e_1 and/or e_2 and/or e_3, we disconnect one or more of the gates g_1, g_2, g_3, and replace their outputs with the values of w_1, w_2, w_3 (which can be assigned at will[20.]). This "enhanced circuit" technique provides us with an opportunity to spot and correct design errors.

Now, the question is: Which are the gates we need to disconnect in order to correct, through a proper sequence of values for rectifying vectors, the overall input/output behavior of the system?

In Example (11) it suffices to set e_2 to true (and the other selection bits to false) and then "fix" the output of the erroneous gate g_2 (using the value of w_2, which is unconstrained). In particular, the 2-step sequence of values computed by g_2 should be 1, 0, and not 1, 1. This fixes the circuit's I/O behavior, and implies that gate g_2 should be a `NOR`.

20. Notice that the transition relation and the error selection vector are represented only once, independently of k, while a different rectifying vector is applied to each of the k steps. So, if the disconnection of a gate is "decided", the gate stays disconnected through all the evolution of the system. This is coherent with the observation that we look for faulty gates, which presumably stay faulty as the system evolves.

We want to be able to automatically find both the candidate faults, and the sequence of rectifying values. This can be obtained by replacing in (27) the transition relation T with the transition relation T_{en} of an enhanced system for which rectification is possible:

$$\begin{aligned}\exists s_0, \ldots, s_k\ \forall t\ \exists s, x, s', w, y.\ [\ & mux_n(s, t, s_0, \ldots, s_{k-1}) \wedge mux_n(s', t, s_1, \ldots, s_k) \wedge \\ & mux_n(x, t, \overline{x_1}, \ldots, \overline{x_k}) \wedge mux_n(y, t, \overline{y_1}, \ldots, \overline{y_k}) \wedge \\ & T_{en}(s, x, e, s', w, y)]\ \wedge\ |e| \leq d\end{aligned} \tag{28}$$

We obtain an open QBF, whose free variables are the bits of the error selector e. For any truth assignment to the bits in e, the formula (28) is a closed QBF which is true only if the enhanced system represented by T_{en} (i.e., the system in which all the gates corresponding to true bits in e have been disconnected and set free to compute arbitrary output values) can comply with the expected I/O behavior. So, each valid assignment to the free variables of (28) identifies a candidate set of faulty gates. There might be many valid assignments to e. As we prefer small redesigns to large ones, we introduce in (28) the additional constraint $|e| \leq d$, which limits the number of bits set to true in e to d. So, the meaning of (28), computed by a QBF solver according to Definition 3.1, is the set of all the ways to fix the system replacing d gates or less. By solving (28) for increasing values of d, starting from 1, we indentify a pool of redesigns of minimal size[21].

Once again, a SAT formulation of the same problem is the most natural alternative to compare with, as it can be attained by the same technique [4]. The main difference is that the transition relation is explicitly replicated along the whole path, so that mesh-securing multiplexers are not necessary. To solve the SAT version, an all-solution solver is employed, and blocking clauses are incrementally added in between runs to prevent the same fix from appearing more than once.

Several random faults have been injected in the six industrial designs discussed in Section 3.2 by manually changing the functionality of certain modules to introduce errors. These faults have been first spotted (through random simulation), then corrected through QBF-based and SAT-based DD techniques. The number of fixes found for some of these cases, and a comparison between the QBF-based and SAT-based time/memory requirements, are shown in Table 5.

To the best of our knowledge, Table 5 depicts the best ever scenario for QBF-based FV. As expected, the QBF formulation is substantially smaller than the SAT-based one (an order of magnitude in our examples), so that it enables us to solve cases whose SAT formulation just does not fit in memory. Furthermore, even when both formulations are affordable, it is sensibly faster (apart from the smallest cases) to solve the QBF-based one than the SAT-based one.

4. Discussion and Conclusion

QBF-based formal verification has been considered a promising approach, essentially because the QBF formalization of many FV tasks is both natural and substantially more

21. The pool of optimal redesigns is then analyzed by human experts, as further considerations concur in choosing what to modify in the actual circuit. It is also possible that some minimal repair cannot be synthesized in practice, because a sequence of rectifying values is requested which no deterministic gate can compute from its inputs.

Table 5. Comparison between QBF and SAT based enumeration of fixes for faulty designs of some industrial benchmarks. The encoding and benchmark set are as described in [55]. The instance "c[X]-e[Y]-v[Z]" corresponds to design error number Y manually inserted in circuit number X (the functionality of one or more gates are arbitrarily changed to produce a faulty circuit) and spotted through counter-example number Z (multiple output sequences may depart from the circuit's expected behavior). Time is measured in seconds. The "Mem." column gives the footprint in megabytes of the file containing the instance. TO stands for timeout (3600 seconds). MO means that either the generator (in case MO is in the *Mem.* column) or the solver (in case it is in the *Time* column) failed for an out-of-memory condition (2 GBytes allotted). The solvers are zChaff v-04.5.13 and sKizzo v-0.10 (the latter using the former as a back-end propositional reasoner).

Circuit 1

		sKizzo / QBF		zChaff / SAT	
instance	#	*Time*	*Mem.*	*Time*	*Mem.*
c1-e1-v1	3	123.2	40	MO	489
c1-e1-v2	3	57.6	21	136.1	242
c1-e1-v3	3	126.5	39	MO	489
c1-e2-v1	1	18.5	11	23.0	118
c1-e2-v2	1	15.5	6.3	10.9	56
c1-e2-v3	1	23.7	11	47.3	242

Circuit 2

		sKizzo / QBF		zChaff / SAT	
instance	#	*Time*	*Mem.*	*Time*	*Mem.*
c2-e1-v1	4	881.4	18	MO	373
c2-e1-v2	6	498.6	9.0	TO	183
c2-e1-v3	4	980.1	18	MO	373
c2-e2-v1	1	20.9	8.6	TO	88
c2-e2-v2	1	20.77	8.6	TO	88
c2-e2-v3	1	20.9	8.6	TO	88

Circuit 3

		sKizzo / QBF		zChaff / SAT	
instance	#	*Time*	*Mem.*	*Time*	*Mem.*
c3-e1-v1	3	3.3	1.2	1.0	4
c3-e1-v2	3	9.1	2.1	2.6	9
c3-e1-v3	3	10.2	2.1	4.6	19
c3-e2-v1	3	TO	62	MO	685
c3-e2-v2	3	1389.6	64	MO	685
c3-e2-v3	3	1115.8	63	MO	685

Circuit 4

		sKizzo / QBF		zChaff / SAT	
instance	#	*Time*	*Mem.*	*Time*	*Mem.*
c4-e1-v1	4	34.3	11	1214.6	125
c4-e1-v2	3	34.8	11	1635.9	125
c4-e1-v3	4	28.9	11	1188.3	125
c4-e2-v1	1	34.7	11	170.7	125
c4-e2-v2	1	36.6	11	172.5	125
c4-e2-v3	1	29.6	11	163.5	125

Circuit 5

		sKizzo / QBF		zChaff / SAT	
instance	#	*Time*	*Mem.*	*Time*	*Mem.*
c5-e1-v1	3	3.8	1.4	2.2	11
c5-e1-v2	3	3.8	1.4	2.2	11
c5-e1-v3	3	3.8	1.4	2.2	11
c5-e2-v1	4	8.3	2.2	10.2	43
c5-e2-v2	4	8.3	2.2	9.3	43
c5-e2-v3	4	7.7	2.2	4.5	43

Circuit 6

		sKizzo / QBF		zChaff / SAT	
instance	#	*Time*	*Mem.*	*Time*	*Mem.*
c6-e1-v1	2	6.6	2.3	38.5	33
c6-e1-v2	2	9.6	2.3	187.5	68
c6-e1-v3	2	6.7	2.3	201.8	68
c6-e2-v1	2	6.6	2.3	34.6	33
c6-e2-v2	2	9.3	2.3	203.5	68
c6-e2-v3	2	6.9	2.3	206.2	68

compact than equivalent ones for contending approaches, i.e., SAT-based ones. Unfortunately, experience with this framework has invariably yielded disappointing performances: Not only QBF fails to improve over SAT, but it seems plainly unable to stand comparison with its opponent.

A range of alternative explanations can be put forward to account for these negative results. Perhaps solvers are just far from their full potential. Or, the problem is in the way FV tasks are formalized. And it cannot be excluded that QBF is simply the wrong formalism to adopt!

This paper aims to contribute to a better understanding and assessment of the situation. Novel experiments and research perspectives were described that shed light on the actual potential of QBF in FV. For the first time, experimental results favorable to QBF were reported. Furthermore, a wide variety of promising research directions were illustrated to shake QBF out of the current impasse.

Our main findings can be summarized as follows. First, "alternative" decision procedures for QBF (i.e., not based on search) seem to have an edge over search-based ones as far as FV tasks are concerned. In most of our experiments (Section 2), the best solvers were not based on search. In particular, we showed how the symbolic skolemization technique, in conjunction with a hybrid inference engine, yields a very competitive reasoner. The success of this alternative paradigm is quite relevant when compared with the evidence that all the state-of-the-art solvers for SAT, and most of those for QBF, embrace the DPLL approach. An analysis of the inference logs of the solver reveals that different problems are best solved via different inference behaviors. This emphasizes the importance of a hybrid reasoning architecture which allows to devise the most effective inference attack on a per-family basis.

Second, it seems that the way problems are encoded does make a large difference to QBF solvers. Universal quantifiers have been usually leveraged for the sake of their syntactic power, i.e., for their capability to compress the problem representation through the syntactic contortions they allow to perform. This is not a sufficient criterion, however, to infer that we are really exercising them as a natural way to state relevant facts or rules, or that we are producing an encoding which is easier to solve for any QBF reasoner. So, on one hand, we can identify classes of problems—for example equivalence checking—which may be best dealt with by SAT solvers, even if their QBF formulation is shorter. On the other hand, alternative QBF encodings for classical problems could substantially improve the response of solvers, as the results in Section 3.2 suggest.

Finally, the main feature of QBF—the alternation of existential and universal quantifications—besides making the language essentially more difficult to deal with, seems to disclose unexpected possibilities for FV tasks. We investigated in Section 3 some of these research directions, like the role of certificates in formal verification, the usage of formulas mentioning free variables, and the strength of quantifiers with restricted span. Some of these approaches resulted in the best ever performances of QBF-based formal verification, compared to SAT.

The conclusion we draw from our investigation is that, despite the negative results reported in the recent literature on the topic, the role of QBF in FV may increase in the near future: Strong "unconventional" QBF solvers start exhibiting reasonable performances, and the need to analyze long (possibly faulty) behaviors of very complex systems demands compact formulations which include universal quantifiers. We believe that QBF, just like

SAT a few years ago, could now start to benefit from a virtuous feedback between "QBF producers" (i.e., researchers interested in solving problem whose natural formulation is in QBF) and "QBF consumers" (i.e., researchers who develop QBF solvers and techniques). We contribute to this effort by deploying public implementations of all the solver versions and advanced techniques described in this paper [16].

Acknowledgments. We thank Arnaud Lallouet for his contributions to improving the paper, Andreas Veneris for his strong encouragement to pursue QBF-based FV and for comments on parts of this work, and the anonymous reviewers for their precious remarks on preliminary versions of the paper.

References

[1] OpenCores's web site, `http://www.opencores.org`, 2006-2007.

[2] The AIGER project: Format, Libraries and Utilities for And-Inverter Graphs, Institute for Formal Models and Verification, Austria, `http://fmv.jku.at/aiger`, 2006.

[3] M. F. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler. Post-Verification Debugging of Hierarchical Designs. In *Proceedings of the International Conference on Computer Aided Design (ICCAD-05)*, 2005.

[4] M. F. Ali, A. Veneris, A. Smith, S. Safarpour, R. Drechsler, and M. S. Abadir. Debugging Sequential Circuits Using Boolean Satisfiability. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD-04)*, 2004.

[5] C. Ansótegui, C. P. Gomes, and B. Selman. The Achilles' Heel of QBF. In *Proceedings of the 20th AAAI National Conference on Artificial Intelligence (AAAI-05)*. AAAI Press, 2005.

[6] G. Audemard and L. Sais. A Symbolic Search Based Approach for Quantified Boolean Formulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, number 3569 in LNCS. Springer, 2005.

[7] A. Ayari and D. Basin. Bounded Model Construction for Monadic Second-order Logics. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV-00)*, number 1855 in LNCS. Springer, 2000.

[8] A. Ayari and D. Basin. QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD-02)*, number 2517 in LNCS. Springer, 2002.

[9] M. Baaz, U. Egly, and A. Leitsch. Normal Form Transformations. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, **1**, chapter 5, pages 273–333. The MIT Press, 2001.

[10] M. Benedetti. sKizzo and ozziKs User Manuals, available at `http://sKizzo.info`, 2006.

[11] M. Benedetti. sKizzo: a QBF Decision Procedure Based on Propositional Skolemization and Symbolic Reasoning. Technical Report 04-11-03, ITC-irst, Center for Scientific and Technological Research, 2004.

[12] M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-04)*, number 3452 in LNCS. Springer, 2005.

[13] M. Benedetti. Extracting Certificates from Quantified Boolean Formulas. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, 2005.

[14] M. Benedetti. Quantifier Trees for QBFs. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, number 3569 in LNCS. Springer, 2005.

[15] M. Benedetti. sKizzo: A Suite to Evaluate and Certify QBFs. In *Proceedings of the 20th International Conference on Automated Deduction (CADE-05)*, number 3632 in LNCS. Springer, 2005.

[16] M. Benedetti. sKizzo's web site, `http://sKizzo.info`, 2005 – 2007.

[17] M. Benedetti. Abstract Branching for Quantified Formulas. In *Proceedings of the 21st AAAI National Conference on Artificial Intelligence (AAAI-06)*. AAAI Press, 2006.

[18] M. Benedetti. Experimenting with QBF-based Formal Verification. In *Proceedings of the 3rd International Workshop on Contraints in Formal Verification (CFV-03), Tallinn*, July 2005.

[19] M. Benedetti, A. Lallouet, and J. Vautard. QCSP Made Practical by Virtue of Restricted Quantification. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2007.

[20] A. Biere. Resolve and Expand. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-04)*, number 3542 in LNCS. Springer, 2004.

[21] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS-99)*, number 1579 in LNCS. Springer, 1999.

[22] L. Bordeaux and E. Monfroy. Beyond NP: Arc-Consistency for Quantified Constraints. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02)*, number 2470 in LNCS. Springer, 2002.

[23] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transaction on Computing*, **35**(8):677–691, 1986.

[24] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Convergence Testing in Term-Level Bounded Model Checking. In *Proceedings of the 12th Conference on Correct Hardware Design and Verification Methods (CHARME-03)*, number 2860 in LNCS. Springer, 2003.

[25] H. K. Büning, M. Karpinski, and A. Flogel. Resolution for Quantified Boolean Formulas. *Information and Computation*, **117**(1):12–18, 1995.

[26] H. K. Büning and T. Lettmann. *Propositional Logic: Deduction and Algorithms.* Cambridge University Press, 1999.

[27] H. K. Büning and X. Zhao. On Models for Quantified Boolean Formulas. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-04)*, number 3542 in LNCS. Springer, 2004.

[28] M. Cadoli, T. Eiter, and G. Gottlob. Default Logic as a Query Language. *IEEE Transactions on Knowledge and Data Engineering*, **9**(3):448–463, 1997.

[29] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 15th National/10th Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. AAAI Press, 1998.

[30] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. *Journal of Automated Reasoning*, **28**(2):101–142, 2002.

[31] P. Chatalic and L. Simon. Multi-Resolution on Compressed Sets of Clauses. In *Proceedings of the 12th International Conference on Tools with Artificial Intelligence (ICTAI-00)*, 2000.

[32] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[33] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In *Proceedings of the 13th Conference on Computer Aided Verification (CAV-01)*, number 2102 in LNCS. Springer, 2001.

[34] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Journal of the ACM*, **5**:394–397, 1962.

[35] N. Dershowitz, Z. Hanna, and J. Katz. Bounded Model Checking with QBF. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, number 3569 in LNCS. Springer, 2005.

[36] N. Een and N. Sorensson. An Extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-03)*, number 2919 in LNCS. Springer, 2003.

[37] U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-03)*, number 2919 in LNCS, 2003.

[38] R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulas. In *Proceedings of the 17th AAAI National Conference on Artificial Intelligence (AAAI-00)*. AAAI Press, 2000.

[39] I. Gent and A. Rowley. Encoding Connect-4 using Quantified Boolean Formulae. Technical Report APES-68-2003, APES Research Group, July 2003.

[40] M. Ghasemzadeh. *A New Algorithm for the Quantified Satisfiability Problem, Based on Zero-Suppressed Binary Decision Diagrams and Memoization.* PhD thesis, University of Potsdam, Germany, 2005.

[41] E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating Search Heuristics and Optimization Techniques in Propositional Satisfiability. In *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR-01)*, number 2083 in LNAI. Springer, 2001.

[42] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formula Satisfiability Library (QBFLIB), `www.qbflib.org`, 2001 − 2007.

[43] E. Giunchiglia, M. Narizzano, and A. Tacchella. QuBE: A system for deciding Quantified Boolean Formulas Satisfiability. In *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR-01)*, number 2083 in LNAI. Springer, 2001.

[44] O. Grumberg, A. Schuster, and A. Yadgar. Memory Efficient All-Solutions SAT Solver and Its Application for Reachability Analysis. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD-04)*, 2004.

[45] T. Jussila and A. Biere. Compressing BMC Encodings with QBF. In *Proceedings of the 4th International Workshop on Bounded Model Checking (BMC-06)*, Seattle (USA), 2006.

[46] J. Katz. *Model Checking with Quantified Boolean Formulas.* PhD thesis, School of Computer Science, Tel-Aviv University, 2005.

[47] J. Katz, Z. Hanna, and N. Dershowitz. Space-Efficient Bounded Model Checking. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE-05)*. IEEE Computer Society, 2005.

[48] J. Kim, J. Whittemore, J. P. M. Silva, and K. A. Sakallah. On Applying Incremental Satisfiability to Delay Fault Problem. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE-00)*, pages 380–384. IEEE Computer Society, 2000.

[49] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. *IEEE Transaction on Computer-Aided Design*, **21**(12):1377–1394, 2002.

[50] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transaction on Computer-Aided Design*, **11**(1):6–22, 1992.

[51] D. Le Berre, M. Narizzano, L. Simon, and A. Tacchella. Second QBF Solvers Evaluation, avaliable on-line at `www.qbflib.org`, 2004.

[52] D. Le Berre, L. Simon, and A. Tacchella. Challenges in the QBF Arena: the SAT'03 Evaluation of QBF Solvers, avaliable on-line at `www.qbflib.org`, 2003.

[53] R. Letz. Advances in Decision Procedures for Quantified Boolean Formulas. In *Proceedings of the First International Workshop on Theory and Applications of Quantified Boolean Formulas (QBF-01)*, Siena (Italy), June 2001.

[54] A.C. Ling, D. P. Singh, and S. D. Brown. FPGA Logic Synthesis Using Quantified Boolean Satisfiability. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, number 3569 in LNCS. Springer, 2005.

[55] H. Mangassarian, A. Veneris, and M. Benedetti. Fault Diagnosis Using Quantified Boolean Formulas. In *Proceedings of the 4th IEEE Silicon Debug and Diagnosis Workshop*, Freiburg (Germany), May 2007.

[56] H. Mangassarian, A. Veneris, S. Safarpour, M. Benedetti, and D. Smith. A Performance-Driven QBF-Based ILA Representation with Applications to Verification, Debug and Test. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD-07)*, 2007.

[57] K. L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, 1993.

[58] K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV-02)*, 2002.

[59] M. Mneimneh and K. Sakallah. Computing Vertex Eccentricity in Exponentially Large Graphs: QBF Formulation and Solution. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-03)*, number 2919 in LNCS. Springer, 2003.

[60] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC-01)*, 2001.

[61] M. Narizzano, L. Pulina, and A. Tacchella. Report of the Third QBF Solvers Evaluation. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:145–164, 2006.

[62] G. Pan and M.Y. Vardi. Search vs. Symbolic Techniques in Satisfiability Solving. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-04)*, number 3542 in LNCS. Springer, 2004.

[63] G. Pan and M.Y. Vardi. Symbolic Decision Procedures for QBF. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP-04)*, number 3258 in LNCS. Springer, 2004.

[64] C. H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[65] D. A. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, **2**(3):293–304, 1986.

[66] J. Rintanen. Construction Conditional Plans by a Theorem-prover. *Journal of Artificial Intelligence Research*, **10**:323–352, 1999.

[67] J. Rintanen. Partial Implicit Unfolding in the Davis-Putnam Procedure for Quantified Boolean Formulae. In *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-01)*, number 2250 in LNCS. Springer, 2001.

[68] S. Coste-Marquis and D. Le Berre and F. Letombe. A Branching Heuristics for Quantified Renamable Horn Formulas. In *Proceedings of 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, number 3569 in LNCS. Springer, 2005.

[69] A. Sabharwal, C. Ansotegui, C. P. Gomes, J. W. Hart, and B. Selman. QBF Modeling: Exploiting Player Symmetry for Simplicity and Efficiency. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-06)*, number 4121 in LNCS. Springer, 2006.

[70] H. Samulowitz and F. Bacchus. Using SAT in QBF. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP-05)*, number 3709 in LNCS. Springer, 2005.

[71] H. Samulowitz and F. Bacchus. Binary Clause Reasoning in QBF. In *Proceedings of 9th Int. Conference on Theory and Applications of Satisfiability Testing (SAT-06)*, number 4121 in LNCS. Springer, 2006.

[72] H. Samulowitz, J. Davies, and F. Bacchus. Preprocessing QBF. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP-06)*, number 4204 in LNCS. Springer, 2006.

[73] S. Sapra, M. Theobald, and E. M. Clarke. SAT-Based Algorithms for Logic Minimization. In *Proceedings of the 21st International Conference on Computer Design (ICCD-03)*, 2003.

[74] W. J. Savitch. Relation Between Nondeterministic and Deterministic Tape Complexity. *Journal of Computer and System Sciences*, **4**, 1970.

[75] C. Scholl and B. Becker. Checking Equivalence for Partial Implementations. In *Proceedings of the 38th Design Automation Conference (DAC-01)*, 2001.

[76] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Proc. of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD-00)*, number 1954 in LNCS. Springer, 2000.

[77] A. Smith, A. Veneris, M. F. Ali, and A. Viglas. Fault Diagnosis and Logic Debugging Using Boolean Satisfiability. *IEEE Transactions in Computer-Aided Design*, **24**(10):1606–1621, 2005.

[78] F. Somenzi. Colorado University Binary Decision Diagrams Package, available on-line at `http://vlsi .colorado.edu /~fabio/CUDD`, 1995.

[79] S. Staber and R. Bloem. Fault Localization and Correction with QBF. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT-07)*, number 4501 in LNCS. Springer, 2007.

[80] L. J. Stockmeyer and A. R. Meyer. Word Problems Requiring Exponential Time. In *Proceedings of the 5th Annual ACM Symposium on the Theory of Computing*, Austin, Texas (USA), 1973.

[81] D. Tang, Y. Yu, D. P. Ranjan, and S. Malik. Analysis of Search Based Algorithms for Satisfiability of Quantified Boolean Formulas Arising from Circuit State Space Diameter Problems. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-04)*, number 3542 in LNCS. Springer, 2004.

[82] C. Thiffault, F. Bacchus, and T. Walsh. Solving Non-clausal Formulas with DPLL search. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT-04)*, number 3542 in LNCS. Springer, 2004.

[83] L. Zhang. Solving QBF with Combined Conjunctive and Disjunctive Normal Form. In *Proceedings of the 21st AAAI National Conference on Artificial Intelligence (AAAI-06)*. AAAI Press, 2006.

[84] L. Zhang and S. Malik. Towards Symmetric Treatment of Conflicts And Satisfaction in Quantified Boolean Satisfiability Solver. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02)*, number 2470 in LNCS. Springer, 2002.

[85] M. Zolda. Comparing Different Prenexing Strategies for Quantified Boolean Formulas. Master's thesis, Institute of Information Systems, University of Vienna, 2005.