

# Dominant Controllability Check Using QBF-Solver and Netlist Optimizer

Tamir Heyman<sup>1</sup>, Dan Smith<sup>2</sup>, Yogesh Mahajan<sup>2</sup>, Lance Leong<sup>2</sup>, and Husam Abu-Haimed<sup>3</sup>

<sup>1</sup> NVIDIA Inc Beaverton, OR, 97006

<sup>2</sup> NVIDIA Inc Santa Clara, CA, 95050

<sup>3</sup> Atrenta Inc Santa Clara, CA, 95050

**Abstract.** This paper presents an application of formal methods to the verification of hardware power management modules. The property being verified is called Dominant Controllability and is a property of a netlist node and a subset of the inputs. The property holds if there exists an assignment to the subset of the inputs such that it sets the node to 0/1 regardless of the values at the rest of the inputs. Verification of power management modules in recent CPU and GPU designs includes hundreds of such properties. Two approaches are described for verifying such properties: netlist optimization and QBF solving. In the latter case, a QBF preprocessor is used, requiring partial model reconstruction. Each method can be used independently or combined into a third algorithm that heuristically selects a method based on its performance on a design. Experimental results for these methods are presented and discussed.

**Keywords:** Controllability · QBF · QBF Preprocessor · Netlist Optimizer

## 1 Introduction

Power management in hardware is becoming a major challenge in recent designs. Power management is significant because the ratio between performance and power consumption is very important for many applications like mobile devices, servers and gaming consoles. The main power consumers in these applications are the CPU and GPU. In order to save power in a very large hardware design, a number of components can be turned on and off by a power management module. This module has a set of inputs that controls a power switch (clamp signal, sleep signal) to each component individually. The design specifications define the set of power switches and a set of inputs that controls them. The challenge is that these switches are controlled by other modules as well. In order for the power module to function correctly, the power module should have dominant control (i.e., have priority) over other modules connected to the switch. Each switch should be turned on and off by an assignment to the power module inputs regardless of any other controlling modules. Existence of such assignments means that the power module has dominant control over the switch.

This paper presents the use of formal methods for checking dominant controllability. This check calls for formal methods due to the large search space. First, the power management module has a large number of inputs that control multiple power switches. Second, many switches are controlled by a large number of control modules other than the power management module. In such cases, random search may fail to find a controlling assignment even when it exists. Neither is enumerating all possible controlling assignments feasible. Formal methods are needed to efficiently verify whether the power module has priority over all the other modules in controlling the power switch.

This paper describes two dominant controllability check methods. The first is called Const-controllability and uses netlist optimization. For each node representing a power switch, this method tries different assignments to the set of specified inputs. For each assignment tried, the check applies fast netlist optimization to identify assignments which force the node to a constant regardless of the other inputs. If such an assignment is found, it is a controlling assignment.

The second method is called QBF-controllability and uses a QBF solver. This method creates a QBF formula which is true if and only if the node is controllable. This check uses a QBF solver that not only returns SAT/UNSAT/ABORT but also returns a controlling assignment for SAT cases. This assignment is used by the netlist optimizer to verify the correctness of the QBF results. This way QBF-controllability utilizes the technology of the QBF solver and the reliability of the netlist optimization. To the best of our knowledge this is the first time QBF is used for this application.

This paper also describes the use of a QBF preprocessor. During our work, we found that using a QBF preprocessor reduces runtimes for QBF solving and reduces the number of ABORT cases. However, additional work was needed to reconstruct the assignment after preprocessing the formula.

Finally, this paper describes Alternating-controllability, an algorithm which uses both QBF-controllability and Const-controllability. In four out of six designs its runtime is close to the best runtime of either method.

## 2 Background

The hardware design is given as a Boolean *netlist* that is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The present work is concerned with combinational netlists only. *Fanin* edges drive a node. A *fanin cone* of a node  $n$  is the subset of all nodes of the netlist reachable transitively from  $n$  through fanin edges. A *primary input* (*PI*) is a node without *fanin* edges.  $PIs(n)$  is the set of all the *PI* nodes in the fanin cone of  $n$ .  $PIs$  is the union of all  $PIs(n)$ . An *assignment*  $A(v_i, p_i)$  forces a value  $v_i \in \{false, true\}$  on a primary input  $p_i$ . An *assignment*  $A(\mathbf{v}, \mathbf{p})$  forces a vector  $\mathbf{v}$  of Boolean values on the same size vector  $\mathbf{p}$  of primary inputs. Note: there are  $2^{|\mathbf{p}|}$  different assignments to  $\mathbf{p}$ . Given an assignment  $A(\mathbf{v}, \mathbf{p})$  and a node  $n$ , we denote by  $val(n, A(\mathbf{v}, \mathbf{p}))$  the value *true*

or *false* of  $n$  under the assignment  $A(\mathbf{v}, \mathbf{p})$ . Note: it is enough to assign a value only to the primary inputs in  $PIs(n)$  for the evaluation of  $n$  to be *false* or *true*.

A *partial* assignment forces values on a subset  $P_1 \subset PIs$  of the primary inputs and leaves the rest of the inputs unassigned. Under such partial assignment  $A(\mathbf{v}_1, \mathbf{p}_1)$ , the value of  $val(n, A(\mathbf{v}_1, \mathbf{p}_1))$  is not always determined. In such cases we say that  $val(n, A(\mathbf{v}_1, \mathbf{p}_1)) = \text{unknown}$ .

### 3 Dominant Controllability

Large hardware designs often include a power management module. This module includes a set  $C$  of distinguished primary inputs that can be set externally by the software which uses the design.  $C$  controls a set  $PS$  of power switches in the design. A power switch may also be controlled by other modules. In order for the power module to function correctly, the power module should have dominant control of the switch over the other modules. This allows the software to manage the power consumption depending on the actual usage.

Part of the hardware specification is the set  $C$  and the set of power switches  $PS$ . A power switch in the design is a node  $n$  in the *netlist* that can be *on* or *off* depending on the node's evaluation. The evaluation of a node  $n$  is only influenced by  $PIs(n)$ . Therefore, for each node  $n \in PS$ , only the subset  $C_n = C \cap PIs(n)$  needs to be analyzed.

Let  $\mathbf{C}_n$  denote a vector of all the elements of  $C_n$  and  $\mathbf{O}_n$  denote a vector of all elements of  $PIs(n) \setminus C_n$ . Recall that  $\oplus$  denotes vector concatenation.

**Definition 1.** Dominant Controllability *A node  $n$  is dominant controllable by  $C_n \subseteq C$  if and only if  $\forall v \exists \mathbf{v} \forall \mathbf{u}. (val(n, A(\mathbf{v}, \mathbf{C}_n) \oplus A(\mathbf{u}, \mathbf{O}_n)) == v)$ , where  $\mathbf{v}$  and  $\mathbf{u}$  are vectors of Boolean variables.*

In other words, the set of inputs  $C_n$  has dominant controllability over  $n$  if there exist two assignments  $A_0 = A(\mathbf{v}_0, \mathbf{C}_n)$  and  $A_1 = A(\mathbf{v}_1, \mathbf{C}_n)$  that set  $n$  to *false* and *true* respectively, regardless of the other inputs' values. In the rest of this paper, we use controllable in the sense of Dominant Controllability.

### 4 Controllability Check Using Netlist Optimization

This approach uses an optimizer which implements fast (linear runtime) netlist optimization algorithms that were introduced for SAT sweeping [5, 10, 13]. Given a node  $n$  and an assignment  $A_1 = A(\mathbf{v}, \mathbf{C}_n)$ , the optimizer returns a reduced cone of  $n$  denoted as  $opt(A_1, n)$ . The reduced netlist  $opt(A_1, n)$  is computed recursively. First, the inputs which appear in  $A_1$  are assigned to corresponding constant values and other inputs are left as free variables. Then every node  $n$  with a fanin  $\langle n_1, n_2 \dots \rangle$  is computed as  $optimizer(n, \langle opt(A, n_1), opt(A, n_2) \dots \rangle)$ . For example if  $n$  is an output of an AND gate with fanins  $\langle n_1, n_2 \rangle$  and  $opt(A_1, n_1) = \text{false}$ , then  $optimizer$  returns the node *false*.

The optimizer has the following property:  $val(opt(A, n), A) \iff val(n, A)$ . Informally, the optimizer does not change the value of the node  $n$ . Let  $\mathbf{C}_n$  be a

vector of all the elements of  $C_n$  and  $\mathbf{O}_n$  be a vector of all elements of  $PIs \setminus C_n$ . In addition let  $\mathbf{F} = A(\mathbf{v}, \mathbf{C}_n) \oplus A(\mathbf{u}, \mathbf{O}_n)$  be a full assignment to  $PIs$ . Then,  $val(opt(A(\mathbf{v}, \mathbf{C}_n), n), A(\mathbf{v}, \mathbf{C}_n)) \iff val(opt(\mathbf{F}, n), \mathbf{F})$ , for any set of values  $\mathbf{u}$  forced on  $\mathbf{O}_n$ . Consequently, if a partial assignment forces a node to a constant value, any partial assignment to the rest of the inputs will not change the node's value. This observation is what enables the use of the optimizer in the Const-controllability check. The algorithm searches for two assignments  $A_0 = A(\mathbf{v}_0, \mathbf{C}_n)$  and  $A_1 = A(\mathbf{v}_1, \mathbf{C}_n)$  which force  $n$  to 0 and 1 respectively.

Const-Controllability checks if a node  $n$  is controllable by the subset  $C_n$ . Its runtime is controlled by a parameter *MaxTries*. The check uses the netlist optimizer to determine if an assignment  $A(\mathbf{v}, \mathbf{C}_n)$  demonstrates controllability. The complexity of the check is linear in the size of  $n$ 's cone, times *MaxTries*. The optimizer does not always return the smallest most optimized cone of  $n$ , since it uses fast linear algorithms and is not NP-complete like SAT. This means that while Const-Controllability is fast, it is incomplete.

Const-controllability search adapts to the size of  $C_n$ . If  $|C_n|$  is very small ( $\leq 5$ ), the algorithm enumerates all possible assignments to  $C_n$ . For larger  $|C_n|$  it first tries two frequently used assignments, all zeros (*Zeros*) and all ones (*Ones*). Our experimental results show that when these assignments demonstrate controllability, Const-controllability runtime is only a few seconds. If  $A_0, A_1$  are still not found after these two assignments, the check continues with randomly picked assignments from the range  $0 \dots 2^{|C_n|} - 1$ .

If Const-Controllability finds both  $A_0$  and  $A_1$ , it terminates, the controllability check passes and returns SAT. If the number of tries reaches *MaxTries* and  $MaxTries < 2^{|C_n|} - 1$  the check terminates and returns  $ABORT_t$ . If all possible assignments  $0 \dots 2^{|C_n|} - 1$  have been checked but  $A_0$  or  $A_1$  have not been found then the check fails and returns  $ABORT_a$ . The difference between  $ABORT_t$  and  $ABORT_a$  is that  $ABORT_t$  results can change with larger *MaxTries*. In contrast,  $ABORT_a$  remains the same for larger *MaxTries*. Note that since the optimizer is incomplete, Const-Controllability returns  $ABORT_a$  and not UNSAT.

Const-Controllability check (presented in Fig. 1) uses two Boolean flags *found<sub>0</sub>* and *found<sub>1</sub>* to indicate whether  $A_0$  and  $A_1$  have been found. The check iterates until it reaches the smaller of *MaxTries*,  $2^{|C_n|} - 1$  or until both *found<sub>0</sub>* and *found<sub>1</sub>* are set. In each iteration, the check sets some values (as discussed previously) on the inputs in  $C_n$  and calls the optimizer. The optimizer analyzes the cone of  $n$  and returns an optimized cone. If the optimized cone of  $n$  is a constant (*false, true*), the check sets the flags *found<sub>0</sub>*, *found<sub>1</sub>* accordingly.

## 5 Controllability Check Using QBF-Solver

QBF-Controllability( $C_n, n, MaxTries$ ) is presented in Fig. 2. It finds each of the assignments  $A_1$  and  $A_0$  in two separate steps. The parameter *MaxTries* is used to control the resources used by the procedure and limits the maximum number of decisions made by the QBF solver. The QBF solver is modified to return the controlling assignment and will be presented in more detail in Section 6.

```

function Const-Controllability ( $C, n, MaxTries$ )
1   $i = 0$  ,  $found_0 = False$ ,  $found_1 = False$ ,  $A_0 = \emptyset$ ,  $A_1 = \emptyset$ 
2  While ( $i < 2^{|C|} \wedge i < MaxTries \wedge (\neg found_0 \vee \neg found_1)$ )
3    If ( $2^{|C|} < MaxTries$ ) Set  $A(i, C)$  //  $i$  is bitvector representation of  $i$ 
4    Else
5      If ( $i = 0$ ) Set  $A(\langle 0, 0, \dots, 0 \rangle, C)$  /* Zeros */
6      If ( $i = 1$ ) Set  $A(\langle 1, 1, \dots, 1 \rangle, C)$  /* Ones */
7      If ( $i > 1$ ) Set  $A(Random(2^{|C|}, C))$ 
8      If  $Eval(\emptyset, opt(A, n)) = False$ 
9         $found_0 = True$ ,  $A_0 = A$ 
10     If  $Eval(\emptyset, opt(A, n)) = True$ 
11        $found_1 = True$ ,  $A_1 = A$ 
12      $i = i + 1$ 
13  If  $found_0 \wedge found_1$  Return ( $SAT, A_0, A_1$ )
14  If  $2^{|C|} < MaxTries$  Return ( $ABORT_a, A_0, A_1$ )
15  Return ( $ABORT_t, A_0, A_1$ )

```

**Fig. 1.** Pseudo-code Const-Controllability

When both  $A_1$  and  $A_0$  are found, the QBF-Controllability check passes and returns SAT. If a proof is found that either assignment does not exist, the check fails and returns UNSAT. In other cases the search terminates inconclusively and returns ABORT. The check is complete (given enough time it returns SAT if and only if the node  $n$  is controllable) assuming the QBF solver itself is complete.

In order to test dominant controllability of node  $n$  to value *true*, the check uses a QBF solver to test if  $\exists v \forall u. (val(n, A(v, C_n) \oplus A(u, O_n)) == true)$  is satisfiable. If the QBF solver returns UNSAT, then we can conclude that there is no controlling assignment. If the QBF solver returns ABORT, we learn nothing.

```

function QBF-Controllability( $C_n, n, MaxTries$ )
1  ( $result, A_1$ ) = QBF-Controllability-Value( $C_n, n, MaxTries, True$ )
2  If  $result == SAT$ 
3    ( $result, A_0$ ) = QBF-Controllability-Value( $C_n, n, MaxTries, False$ )
4  Return ( $result, A_1, A_0$ )

```

**Fig. 2.** Pseudo-code QBF-Controllability

The function QBF-Controllability-Value( $C_n, n, MaxTries, val$ ) is described in Fig. 3. It searches for a controlling assignment to the inputs in  $C_n$  that sets the node  $n$  to the value *val*. First, it calls GenQDIMACS( $F$ ) to translate the formula  $F$  which is a quantified netlist expression to a QDIMACS [1] formatted formula  $F_{qbf}$ . The generated QDIMACS instance is of the form  $F_{qbf} = \exists V_0 \forall V_1 \exists V_2. CNF(n)$  where  $V_0$  maps to  $C_n$ ,  $V_1$  maps to  $PIs(n) \setminus C_n$  and  $V_2$  has variables that do not correspond to primary inputs. The GenQDIMACS(...)

procedure has some implementation details which are outside the scope of this paper: it bitblasts the netlist, runs constant propagation and does some simple bitlevel optimizations during the translation.

Since we want to call the QBF solver on  $F_{qbf}$ , and then use the partial model returned by the QBF solver to infer an assignment to primary inputs in  $C_n$ ,  $\text{GenQDIMACS}(\dots)$  also returns a one-one mapping  $M_{qbf} : C_n \rightarrow V_0$ . If a variable  $v \in V_0$  is assigned to *true* by the QBF solver, then the corresponding primary input  $p = M_{qbf}^{-1}(v)$  is assigned to *true*, and so on.<sup>4</sup>

$\text{QBF-Solver}(\dots)$  returns SAT, UNSAT or ABORT. If  $F_{qbf}$  is SAT, the QBF solver also returns an existential prefix  $A_{outer-\exists}$  of the satisfying model. In this case, the optimizer evaluates  $\text{opt}(A, n)$  to verify the returned controlling assignment. This verification step guards against bugs in the translation process from quantified netlist expression to QDIMACS and bugs in the QBF solver itself.

function **QBF-Controllability-Value** ( $C_n, n, \text{MaxTries}, \text{value}$ )

```

1   $F = \exists v \forall u. (val(n, A(v, C_n) \oplus A(u, O_n)) == value)$ 
2   $(F_{qbf}, C_{qbf}) = \text{GenQDIMACS}(F)$ 
3   $(result, A_{outer-\exists}) = \text{QBF-Solver}(F_{qbf}, \text{MaxTries})$ 
4   $A = \emptyset$ 
5  If  $(result == \text{SAT})$ 
6    foreach  $(val, var\_id) \in A_{outer-\exists}$ 
7       $A = A \cup (val, M_{qbf}^{-1}(var\_id))$ 
8    If  $(\text{opt}(A, n) \neq val)$ 
9       $result = \text{ABORT}$ 
10 Return  $(result, A)$ 
```

**Fig. 3.** Pseudo-code QBF-Controllability-Value

## 6 QBF-Solver Providing Existential Prefix Models

For our application, we need to check the satisfiability of the formula  $F = \exists S_0 \forall S_1 \exists S_2. \phi_{cnf}$  and if a satisfying model exists, we must return a Boolean assignment to all the variables in  $S_0$  which can be extended to a model of  $F$ . (It's possible that a subset of  $S_0$  is adequate for purposes of controllability, but we ignore that for sake of simplicity.)

The formula  $\exists S_0 \forall S_1 \exists S_2. \phi_{cnf}$  is a quantified Boolean formula (QBF) in *prenex* form, with  $\exists S_0 \forall S_1 \exists S_2$  being the quantifier prefix of  $F$  while propositional formula  $\phi_{cnf}$  forms the *matrix* of  $F$ . This formula is *closed*, with  $S_0, S_1, S_2$  being a partition of the set of all the Boolean variables that occur in  $\phi_{cnf}$ . The variables in  $S_i$  happen to be at scope  $i$ . Here, scope 0 is the *outermost* scope, while scope

<sup>4</sup> The mapping  $M_{qbf}$  is used by  $\text{GenQDIMACS}(\dots)$  during construction of  $F_{qbf}$  to add the quantifier prefix portion of the QDIMACS formula.

2 is the *innermost* scope. (While scanning the quantifier prefix from left to right, the scope increases whenever the quantifier type changes from *existential*  $\exists$  to *universal*  $\forall$  or vice versa.) The concepts of scope and *nesting* of scopes are useful for interpreting QBF formulas, and this can be done recursively. We illustrate this informally here for prenex formulas. If  $Q = \exists v.Q'$  ( $Q = \forall v.Q'$ ) is a closed QBF formula and  $v$  does not occur in  $Q'$ , then  $Q$  is satisfiable iff  $Q'$  is satisfiable. If  $v$  does occur in  $Q'$ , then let  $Q'|_{v=0}$  ( $Q'|_{v=1}$ ) denote the result of *substituting* 0 (1) for the *free* occurrence of  $v$  in  $Q'$ . Then, if  $Q = \exists v.Q'$  ( $Q = \forall v.Q'$ ) is a closed QBF formula and  $v$  occurs in  $Q'$ , we say that  $Q$  is satisfiable iff either (both) of  $Q'|_{v=0}$  and  $Q'|_{v=1}$  are satisfiable. Lastly, note that both  $Q'|_{v=0}$  and  $Q'|_{v=1}$  are closed prenex QBF formulas and the procedure can be repeated.

The above informal description of the semantics suggests the existence of *tree-like models* which witness the satisfiability of a closed prenex QBF formula. A tree-like model is a tree where the child nodes are variable assignments, and its structure is such that it demonstrates why the formula is satisfiable. In order to do so, a tree-like model satisfies the following constraints. First, the variable assignments along any path from the root to any leaf must form a satisfying assignment for the matrix  $\phi_{cnf}$ . Second, any node corresponding to an assignment to a universal variable must have a sibling with the same variable assigned to the opposite value. Third, the scope of the variable associated with a node must be  $\leq$  the scope of variables associated with all its descendants.

A more detailed presentation of preliminaries of QBF formulas, including scopes, may be found in [11]. Tree-like models are introduced and discussed in more detail in [15].

In this paper, when we use the word *model*, we have tree-like models in mind. We now introduce the concept of existential prefix model, as motivated by our application.

**Definition 2.** Existential Prefix Model *If  $F = \exists S_0 \forall S_1 \exists S_2. \phi_{cnf}$ , then we define an existential prefix model for  $F$  to be any assignment  $A$  to all variables in  $S_0$  which can be extended to a model for  $F$ . We use the notation  $EPM(A, F)$  to denote that  $A$  is an existential prefix model for  $F$ .*

Our QBF solver provides existential prefix models for QDIMACS instances. It combines a search/learning based QBF solver (**depqbf-1.0** [11]) with a modified QBF preprocessor (**bloqger** [3]) while maintaining the necessary bookkeeping information to calculate existential prefix models in case of a SAT result.

**bloqger** is a QDIMACS preprocessor that transforms formula  $F$  to  $F'$  while preserving satisfiability. It does not guarantee that a model for  $F'$  is a model for  $F$ . In case  $F'$  is UNSAT, there is no existential prefix model for  $F$  because **bloqger** preserves satisfiability. In case  $F'$  is SAT, then we can use an existential prefix model for  $F'$  together with knowledge of the transformations applied by the preprocessor to reconstruct an existential prefix model for  $F$ . We view the execution of **bloqger** as a sequence of transformations of formulas  $F = Q_0 \rightarrow Q_1 \rightarrow \dots \rightarrow Q_{n-1} \rightarrow Q_n = F'$ . As the transformations are applied, information needed to reconstruct the existential prefix model is pushed onto a

stack. **depqbf** is then called on  $F'$ . If **depqbf** declares SAT, the reconstruction procedure queries the solver state in **depqbf** (collects the most recent assignments to variables in  $S_0$ ) to determine an existential prefix model for  $F' = Q_n$ . With that initial seed, the reconstruction procedure then pops the reconstruction information from the stack to reconstruct an existential prefix model for  $Q_{n-1}$  and continues in this fashion until the stack is empty and an existential prefix model for  $F$  is determined.

The key observation about **depqbf** is that when it declares a SAT result for  $F = \exists S_0 \forall S_1 \exists S_2. \phi_{cnf}$ , it retains the most recent assignments to  $S_0$ , and these assignments can be extended to a model for  $F$ . Put differently, the solver terminates with SAT result when it derives a satisfied cube under the most recent assignment to  $S_0$ . Note that the solver maintains the most recent assignments to variables that have been eliminated from the matrix, for example by applications of pure literal rule or unit clause rule. As a result, no modifications were needed to **depqbf** itself since its API provides the necessary functions to inspect solver state.

The modifications needed to **bloqger** were more extensive, and are related to partial model reconstruction. In the following, we review individual transformations  $Q_{pre} \rightarrow Q_{post}$  applied by **bloqger** and summarize the information that needs to be saved in order to reconstruct the existential prefix model. Note that the initial QBF formula is in prenex form with the matrix in CNF form, and each transformation applied by **bloqger** maintains this form. To make the presentation easier, we introduce the following notation.

**Definition 3.** *Outer(C,l) & Inner(C,l): Given a clause  $C$  and a literal  $l \in C$ , we use the notation  $Outer(C, l)$  for the set of all literals in  $(C \setminus l)$  which have a scope outside ( $\leq$ ) the scope of  $l$ . The notation  $Inner(C, l)$  denotes  $C \setminus (Outer(C, l) \cup l)$ .*

For example, in  $\exists x \forall y \exists z. C_1 \wedge C_2$ , where  $C_1 = \{x, y, z\}$  and  $C_2 = \{y, z\}$ , we have  $Outer(C_1, z) = \{x, y\}$  and  $Inner(C_2, z) = \{y\}$ . Note that we treat the clause  $C$  as either a disjunction of literals or a set of literals, based on the context.

1. *Unused Variable Removal:* A variable which is never used in  $\phi_{cnf}$  can be dropped from the quantifier list. No information needs to be saved for reconstruction. If an unassigned variable is encountered during any reconstruction steps, we assign it to a concrete value.<sup>5</sup>
2. *Pure Literal Rule:* If an existential (universal) variable  $v$  occurs in  $\phi_{cnf}$  in only one phase as literal  $l$ , then all clauses containing  $l$  (occurrences of  $l$ ) can be removed from  $\phi_{cnf}$ . If  $v \in S_0$ , we push the constraint  $l$  on the stack.
3. *Forall Reduction:* If the literal  $l$  with the greatest scope in a clause  $C$  is universal, then  $C$  can be replaced by  $(C \setminus l)$ . This is model preserving and hence no reconstruction is needed.
4. *Unit Clause Rule:* If a forall reduced clause  $C$  has only one (existential) literal  $l$ , then all occurrences of  $\neg l$  are removed from  $\phi_{cnf}$ . This is model preserving. (It is followed by pure literal rule application.)

<sup>5</sup> If we don't assign a concrete value, the next time we see the variable during reconstruction, we may interpret the variable inconsistently.

5. *Subsumption Based Clause Deletion*: If clauses  $C_1, C_2 \in \phi_{cnf}$  are such that  $C_2 \subset C_1$ , then clause  $C_1$  can be deleted. This is model preserving.
6. *Hidden Literal Addition*: For clauses  $C_1, C_2 \in \phi_{cnf}$ , if there is a literal  $l \in C_2$  such that  $(C_2 \setminus l) \subset C_1$ , then  $C_1$  can be replaced with  $(C_1 \cup \neg l)$ . Since  $C_1 \wedge C_2 = (C_1 \cup \neg l) \wedge C_2$  under these conditions, this is model preserving.
7. *Forward/Backward Clause Strengthening*: This is the reverse direction of hidden literal addition and is model preserving.<sup>6</sup>
8. *Tautology Removal*: If a clause  $C \in \phi_{cnf}$  contains both a literal  $l$  and its negation  $\neg l$ , then  $C$  can be deleted. This is model preserving.
9. *Covered Literal Addition*: With  $l$  being an existential literal, define  $N_l$  as being the set  $\{C' \in \phi_{cnf} \mid \neg l \in C'\}$ . Any clause  $C \in \phi_{cnf}$  containing the existential literal  $l$  can then be replaced with  $C \cup N_l^{outer}$ , where  $N_l^{outer} = \{\bigcap_{C' \in N_l} Outer(C', \neg l)\}$ .  
Given a model<sup>7</sup> for  $Q_{post}$ , consider the subtrees of the model rooted at occurrences of  $var(l)$ . For each such subtree, the path from the root of the model to the root of the subtree contains assignments to all the variables in  $Outer(C, l) \cup N_l^{outer}$ . If  $N_l^{outer}$  is true under these assignments, then set  $var(l)$  at the root of this subtree such that  $l$  is *true* and make  $C$  satisfied along all paths through the subtree. If  $N_l^{outer}$  is *false* under these assignments, then  $Outer(C, l)$  and hence  $C$  must already be *true* along all paths of the model through this subtree. This process gives a model for  $Q_{pre}$ . For reconstruction of  $EPM(Q_{pre})$ , we push the constraint  $N_l^{outer} \Rightarrow l$  on the stack when  $var(l) \in S_0$ .
10. *Blocked Clause Elimination*: With  $l \in C$  being an existential literal, define  $N_l$  as being the set  $\{C' \in \phi_{cnf} \mid \neg l \in C'\}$ . If  $Outer(C, l) \cup Outer(C', \neg l)$  is a tautology for each  $C' \in N$ , then the clause  $C$  can be deleted.  
Given a model for  $Q_{post}$ , consider the subtrees of the model rooted at occurrences of  $var(l)$ . For each such subtree, the path from the root of the model to the root of the subtree contains assignments to all the variables in  $Outer(C, l)$ . If  $Outer(C, l)$  evaluates to *false* under this assignment, then set  $var(l)$  such that  $l$  is *true*. Note that all paths through the subtree also satisfy all clauses in  $N_l$ . This process gives a model for  $Q_{pre}$ . Since we are interested in reconstructing  $EPM(Q_{pre})$ , we push the constraint  $\neg Outer(C, l) \Rightarrow l$  on the stack when  $var(l) \in S_0$ .
11. *Hidden Tautology Removal*: This is a sequential application of one or more steps of hidden literal addition and covered literal addition, followed by tautology removal. Reconstruction information is pushed on the stack for each (speculative) covered literal addition during the hidden tautology check. If the clause is not found to be redundant, the speculative additions to the reconstruction stack are popped, though this is not needed for correctness.
12. *Hidden Blocked Clause Elimination*: This is a sequential application of one or more steps of hidden literal addition and covered literal addition, fol-

<sup>6</sup> “forward” and “backward” refer to the relative chronological ages of the clauses involved. There are separate procedures in **bloqger** for each direction.

<sup>7</sup> We really want a model such that all children of  $var(l)$  have a strictly deeper scope. We get this by reordering variables in the same scope as  $var(l)$  in a model for  $Q_{post}$ .

lowed by blocked clause elimination. Similar to the case of hidden tautology removal, reconstruction information for covered literal additions is pushed speculatively on to the stack. If blocked clause elimination succeeds, then the reconstruction information for the blocked clause elimination is also pushed on the stack. Otherwise, the speculative additions to the stack are popped, though this is not needed for correctness.

13. *Equivalent literal removal*: If  $\phi_{cnf} \Rightarrow (x == y)$ , the scope of  $var(x)$  is outside ( $\leq$ ) the scope of  $var(y)$ , and  $var(y)$  is an existential variable, then the literal substitutions  $[y \mapsto x]$  and  $[\neg y \mapsto \neg x]$  are applied to  $\phi_{cnf}$ . If  $var(y) \in S_0$ , then the constraints  $x \Rightarrow y$  and  $\neg x \Rightarrow \neg y$  are pushed on the stack.
14. *Existential Variable Elimination*: With  $l$  being an existential literal, define  $P_l$  as being the set  $\{C \in \phi_{cnf} \mid l \in C\}$  and  $N_l$  as being the set  $\{C \in \phi_{cnf} \mid \neg l \in C\}$ . Suppose that for each  $(C_p, C_n) \in P_l \times N_l$ , either  $(Outer(C_p, l) \cup Outer(C_n, \neg l))$  is a tautology or  $(Inner(C_p, l) \cup Inner(C_n, \neg l))$  is the empty set. Under these conditions,  $P_l \cup N_l$  can be deleted from  $\phi_{cnf}$  and replaced with  $R = \{(C_p \setminus l \cup (C_n \setminus \neg l)) \mid (C_p, C_n) \in P_l \times N_l\}$ .  
Given a model for  $Q_{post}$ , consider the subtrees rooted at occurrences of  $var(l)$ . For each such subtree, the assignments along the paths through the subtree satisfy all clauses in  $R$ . Let  $A$  be the partial assignment implied by the path from the root of the model to the root of the subtree. If  $Outer(C_p, l)$  is false under  $A$  for any of the  $C_p \in P_l$ , then modify the assignment to  $var(l)$  at the root of this subtree such that  $l$  is *true*. Otherwise, modify the assignment to  $var(l)$  at the root of this subtree such that  $l$  is *false*. This makes all paths through the subtree satisfy all clauses in  $P_l \cup N_l$ . This process gives a model for  $Q_{pre}$ . If  $var(l) \in S_0$ , push the constraint  $(\bigwedge_{C_p \in P_l} Outer(C_p, l)) \Leftrightarrow \neg l$  on the stack.
15. *Forall Variable Expansion*: This transforms formula  $Q_{pre} = \exists S_0 \forall S_1 \exists S_2. \phi_{cnf}$  to  $Q_{post} = \exists S_0 \forall (S_1 \setminus v) \exists (S_2 \cup S_2^{copy}). \phi_{cnf} [v \mapsto 1] \wedge \phi_{cnf} [v \mapsto 0, S_2 \mapsto S_2^{copy}]$  where  $v \in S_1$  and  $S_2^{copy}$  is a fresh set of newly introduced existential variables. An existential prefix model for  $Q_{post}$  restricted to  $S_0$  is an existential prefix model for  $Q_{pre}$  and no reconstruction information needs to be saved.
16. *Clause Splitting*: Clause splitting is used by the preprocessor to split long clauses into shorter clauses by introducing new existential variables at the deepest scope of the formula. Given a model for  $Q_{post}$ , deleting the part corresponding to the newly added variables gives a model for  $Q_{pre}$ .

To summarize, we need to push reconstruction information on the stack for the following transformations - *Pure Literal Rule*, *Covered Literal Addition*, *Blocked Clause Elimination*, *Equivalent Literal Removal* and *Existential Variable Elimination*.

Finally, we conclude the section with a note on how we tested the QBF solver implementation. We used a fuzzer (Qbfuzz) [4] to generate a small random QDIMACS instance  $F$  with 30–100 variables. In case the QBF solver returns UNSAT on  $F$ , the test harness confirms that `depqbf` does not return SAT on  $F$ . In case the QBF solver returns SAT and an existential prefix model  $A$ , the

test harness confirms that `depqbf` returns SAT on  $F''$  where  $F''$  is  $F$  with unit clauses added to force the assignment  $A$ .

## 7 Controllability Check Using an Alternating Approach

Alternating-Controllability( $C, PS$ ) depicted below in Figure 4 checks that each power switch node in the set  $PS$  is controllable by a set of primary inputs  $C$ . This algorithm harnesses the benefits of the two methods: QBF-Controllability and Const-Controllability. It runs two rounds: a fast round with a time budget  $TimeLimit$  for each method. Based on the winner of the first round, a second round is executed without a time limit ( $None$ ).

In the first round, using Const-Controllability first is better because QBF-Controllability has a higher overhead. The first round limits the number of tries per node to  $FastTries$  to avoid spending all the method's time budget on a single node. In the second round the order in which the methods are run is determined by the results of the first round. In this round there is no time limit. In addition a larger number of tries  $MaxTries$  is used per node by each method.

The algorithm maintains the set  $PS_i$  of nodes left to verify. A node is removed from the set  $PS_i$  when one of the methods returns SAT or UNSAT for it. The algorithm reports ABORT for the remaining nodes in this set after the two rounds end.

```

function Alternating-Controllability( $C, PS$ )
1   $Order = Const\_Controllability, QBF\_Controllability$ 
2   $PS_i = PS$ 
3  Foreach method in the sequence  $Order$ 
4     $D[method] = Timed-Controllability(method, C, PS_i, FastTries, TimeLimit)$ 
5     $PS_i = PS \setminus D[method]$ 
6  If  $|D[Const\_Controllability]| < |D[QBF\_Controllability]|$ 
7     $Order = QBF\_Controllability, Const\_Controllability$ 
8  Foreach method in the sequence  $Order$ 
9     $D[method] = Timed-Controllability(method, C, PS_i, MaxTries, None)$ 
10    $PS_i = PS \setminus D[method]$ 
11  Report ABORT for all nodes in  $PS_i$ 

```

**Fig. 4.** Pseudo-code Alternating-Controllability

$Timed-Controllability(method, C, PS, MaxTries, TimeLimit)$  depicted in Figure 5 runs Const-Controllability or QBF-Controllability with a time budget. It returns all the nodes that are found to be SAT or UNSAT.

## 8 Related Work

Controllability is a property of a node in the netlist [2]. In combinational logic, controllability is the ability to set a node to a specified value by an assignment

```

function Timed-Controllability(method, C, PS, MaxTries, TimeLimit)
1  Iterate n over all PS or until runtime > TimeLimit
2    If method = Const-Controllability
3      Const-Controllability(C, n, MaxTries)
4    Else
5      QBF-Controllability(C, n, MaxTries)
6  Return all nodes n ∈ PS that resulted in SAT or UNSAT

```

**Fig. 5.** Pseudo-code Timed-Controllability

to the primary inputs. In Automatic Test Generation (ATG), controllability is a measure of the difficulty to set each signal to *true* and *false*. A precise measure of such controllability is hard to define/compute and random controllability is used for some applications. Random controllability looks at the probability of setting a signal to *true* or *false* using random assignments to the primary inputs. Our work focuses on a specific concept : Dominant-controllability. This notion of controllability emphasizes the element of dominance of a subset of inputs over other inputs to the controllability concept. In recent complicated designs such as GPU and CPU the specification of the power control module includes the dominance element in order to allow the software running the design to control the power consumption. Our work applies formal methods to verify Dominant-controllability.

Three valued simulation can also be used for Const-Controllability instead of the optimizer. If the value of *n* is a constant, then the requested assignment to *C<sub>n</sub>* is found. However, three valued simulation may be too pessimistic (the value of *n* is X, even though it should be a constant) and may result in missing a valid assignment. For example, an AND gate with two inputs *In* and  $\neg In$  may evaluate to X when it is actually *false*. In addition, when the number of inputs in *C<sub>n</sub>* is very large it is not practical to enumerate all possibilities.

In our work, we use the QBF solver **depqbf** and the QBF preprocessor **bloqqr**. For search/learning based QBF solvers such as **depqbf**, there is work related to certificate generation for QBF results [14, 16]. There is also related work that involves model/proof reconstruction after steps including preprocessing, such as [7, 8, 12]. These efforts are aimed at a more general setting than our application, which is limited to the significantly simpler problem of reconstructing existential prefix models for satisfiable QBF formulas. Specifically, we don't need to generate a full certificate since we are only interested in an existential prefix model. Our verification of SAT results is done using a constant propagation and optimization process which is very fast and efficient. Since we accept ABORT results for our application, we did not feel the cost of certificate generation and verification was justified for the case of UNSAT results. Since our work is more focused on existential prefix model reconstruction after preprocessing by **bloqqr**, we would also like to point to what we feel is the closest related work which is actually for solution reconstruction after preprocessing of CNF formulas in the domain of SAT solvers [9]. We also note that some of the specific

techniques used within **bloqqer** have been introduced in other preprocessors, for example, existential variable elimination in [6], and analogous procedures for existential prefix model reconstruction should apply.

## 9 Experimental Results

Table 1 presents the experimental results for Const-Controllability on six industrial designs. *D1*, *D2*, *D3* and *D4* are previously verified designs with no known bugs. *D5* and *D6* are newer designs with potential bugs. Each design specifies a control set  $C$ . Const-Controllability checks Dominant Controllability of each node (power switch) in the power management module. We controlled the runtime of Const-Controllability by limiting *MaxTries* to 100 for each node.

The SAT column shows the number of controllable nodes. *ABORT<sub>a</sub>* shows the number of nodes that are not controllable even after the algorithm tests all  $2^{|C|}$  possible assignments. The *ABORT<sub>t</sub>* column shows the number of nodes for which the algorithm found only one or none of the necessary assignments ( $A_0$  or  $A_1$ ) before hitting the threshold of 100 random assignments. The *Zeros* (*Ones*) assignment sets all inputs to 0 (1). *1stRand* is the first random assignment tried. The columns *Zeros*, *Ones* and *1stRand* show how many times these assignments were the controlling assignments  $A_0$  or  $A_1$ . The column  $\widetilde{PIs}(n)$  is the median size of  $PIs(n)$ , with the minimum and maximum values in parentheses. The column  $\widetilde{C}_n$  is the median size of  $C_n$ , with the minimum and maximum values in parentheses. The averages are slightly higher, but the median represents the strength of the algorithm on the smaller examples. The last column is the total runtime for checking all nodes in seconds. Note: Since each node requires two assignments, the sum of *Zeros*, *Ones* and *1stRand* may be larger than the total number of nodes.

Our first observation is that the performance of Const-Controllability improves as the percentage of uncontrollable nodes decreases. The reason is that Const-Controllability is unable to conclude that a node is uncontrollable. Therefore, each node that is uncontrollable consumes runtime without contributing to the number of nodes resolved.

The second observation is that *Ones* assignment is very useful in some designs like *D3* where almost all of the power switches tend to use this assignment! (The *Ones* assignment forces some nodes to 1 and others to 0).

The third observation is that many of the assignments are easy to guess. In *D3*, the first random assignment forces the node to 1 or 0 in 93% of the cases.

Last observation is that in newer designs the size of the controlling set is very small. For example in *D5* the median size of  $C_n$  is only seven – for nodes with six control inputs, the algorithm terminates after enumerating all 64 assignments.

Table 2 presents the experimental results of QBF-Controllability. The table shows the results for the same six designs as Table 1. In the modified QBF solver, we run **bloqqer** preprocessor with no time limit and limit the number of decisions made by **depqbf** to 3,000. We tested all the SAT results using our

**Table 1.** Const-Controllability

|    | <i>SAT</i> | <i>ABORT<sub>a</sub></i> | <i>ABORT<sub>t</sub></i> | <i>Zeros</i> | <i>Ones</i> | <i>1stRand</i> | $\widetilde{PIs}(n)$ | $\widetilde{C}_n$ | <i>Runtime</i> |
|----|------------|--------------------------|--------------------------|--------------|-------------|----------------|----------------------|-------------------|----------------|
| D1 | 538        | 0                        | 9                        | 0            | 538         | 93             | 53(53–108)           | 33(33–74)         | 178            |
| D2 | 195        | 0                        | 0                        | 0            | 195         | 76             | 30(30–280)           | 21(21–154)        | 126            |
| D3 | 4,518      | 0                        | 0                        | 0            | 4,517       | 4,203          | 94(58–305)           | 58(37–98)         | 12,044         |
| D4 | 508        | 0                        | 26                       | 0            | 488         | 272            | 88(34–449)           | 59(25–236)        | 13,226         |
| D5 | 0          | 0                        | 805                      | 1            | 0           | 0              | 36(7–74)             | 7(2–46)           | 147,506        |
| D6 | 0          | 114                      | 293                      | 1            | 0           | 0              | 73(15–98)            | 34(2–61)          | 64,549         |

optimizer and all of them were verified. We choose to limit the runtime using number of decisions to get deterministic results.

In all six designs, using the preprocessor reduced the total runtime and the number of timeouts. For example, without the preprocessor *D1* has 30 timeouts and *D2* has 27 timeouts.

**Table 2.** QBF-Controllability

|    | <i>SAT</i> | <i>UNSAT</i> | <i>ABORT</i> | <i>#clauses</i> | $ \exists $ | $ \forall $   | <i>Runtime</i> |
|----|------------|--------------|--------------|-----------------|-------------|---------------|----------------|
| D1 | 547        | 0            | 0            | 592(489–2,435)  | 39(33–74)   | 23(19–31)     | 755            |
| D2 | 194        | 0            | 1            | 667(174–4,005)  | 36(21–154)  | 23(9–124)     | 356            |
| D3 | 3,810      | 0            | 708          | 1k(0.9k–4k)     | 64(37–98)   | 60(32–203)    | 24,215         |
| D4 | 352        | 0            | 182          | 1.5k(0.2k–7.5k) | 46(25–236)  | 79(9–402)     | 3,281          |
| D5 | 0          | 789          | 16           | 7k(0.4k–244k)   | 13(2–46)    | 281(16–9,645) | 27,347         |
| D6 | 0          | 399          | 8            | 4k(0.4k–111k)   | 14(2–61)    | 217(16–4,854) | 8,818          |

The columns *SAT*, *UNSAT* and *ABORT* show the distribution of QBF-solver results. *#clauses* is the average number of clauses for each example. The column  $|\exists|$  has the average number of variables in the outermost existential quantifier, with the min and max values in parentheses. The column  $|\forall|$  has the average number of variables in the outermost universal quantifier, with the min and max values in parentheses. *Runtime* is the total runtime of checking all the nodes in seconds.

Our first observation is that the performance of QBF-Controllability improves as the percentage of bugs increases. In these cases, the solver quickly realizes that there is no way to satisfy the formula and returns UNSAT.

Our second observation is that on simple cases, QBF-Controllability is slower than Const-Controllability. There are a few possibilities which explain this. First is that Const-controllability is using the *Ones* assignment which has a high rate of success on these designs. Second, the QBF-Controllability makes two calls to the modified solver per node, while Const-Controllability runs a single loop to look for both assignments. Lastly, QBF-Controllability incurs an overhead for

generating the QDIMACS instance while Const-Controllability operates natively on the netlist.

Table 3 presents the experimental results of Alternating-Controllability. The table shows the results for the same six designs as Table 1. The columns *SAT*, *UNSAT* and *ABORT* show the distribution of the final results at the end of both rounds. The *FirstRound* and *SecondRound* columns show the number of SAT and UNSAT results found by QBF-Controllability and Const-controllability during each round. *Runtime* is the total runtime in seconds for checking all the nodes.

Our first observation is that the winner in the first round is indeed the faster method for all designs. For example, Table 1 and Table 2 show that *D1* is faster when using Const-controllability compared to QBF-controllability, and Const-controllability is indeed the winner of the first round in this case.

Our second observation is that the total runtime of Alternating-Controllability is closer to the best of Const-Controllability and QBF-Controllability than to the sum of them. For example the runtime for *D5* is 67k seconds. It is closer to the best runtime 27k seconds (QBF-Controllability) than to the sum (174k).

**Table 3.** Alternating-Controllability

|    | <i>SAT</i> | <i>UNSAT</i> | <i>ABORT</i> | First Round |       | Second Round |       | <i>Runtime</i> |
|----|------------|--------------|--------------|-------------|-------|--------------|-------|----------------|
|    |            |              |              | QBF         | Const | QBF          | Const |                |
| D1 | 547        | 0            | 0            | 9           | 538   | 0            | 0     | 262            |
| D2 | 195        | 0            | 0            | 1           | 194   | 0            | 0     | 157            |
| D3 | 4,518      | 0            | 0            | 68          | 92    | 0            | 4,358 | 13,221         |
| D4 | 511        | 0            | 26           | 0           | 14    | 0            | 497   | 15,115         |
| D5 | 0          | 789          | 16           | 10          | 0     | 779          | 0     | 67,660         |
| D6 | 0          | 399          | 8            | 19          | 0     | 280          | 0     | 15,292         |

## 10 Future Work

A promising future direction is to use a SAT solver instead of the netlist optimizer in Const-Controllability. When testing if a node  $n$  is forced to *true* (*false*) by a partial assignment to the set  $C_n$  we can query a SAT solver to check if there is any assignment to the rest of the free inputs in  $PIs(n)$  which sets  $n$  to *false* (*true*). If the SAT solver returns UNSAT then we will conclude that  $n$  is forced by this assignment to *true* (*false*). Since the SAT instances for each node will be closely related we will look into using an incremental SAT solver.

Another future direction is to improve the resource control and reduce the overhead of the translation from netlist to QDIMACS. A circuit based QBF solver which is capable of providing existential prefix models can avoid the translation overhead. Another possibility is to evaluate the performance of QBF solvers which can utilize non-prenex and non-CNF formula representations.

## References

1. QDIMACS Standard Version 1.1 (2005) (Released on December 21 2005) (2005)
2. Abramovici, M., Breuer, M.A., Friedman, A.D.: Digital Systems Testing and Testable Design. Computer Science Press (1990)
3. Biere, A., Lonsing, F., Seidl, M.: Blocked Clause Elimination for QBF. In: Proceedings of the 23rd International Conference on Automated Deduction. pp. 101–115. CADE’11, Springer-Verlag, Berlin, Heidelberg (2011)
4. Brummayer, R., Lonsing, F., Biere, A.: Automated Testing and Debugging of SAT and QBF Solvers. In: Theory and Applications of Satisfiability Testing–SAT 2010, pp. 44–57. Springer (2010)
5. Darringer, J.A., Jr., W.H.J., Berman, C.L., Trevillyan, L.: Logic Synthesis Through Local Transformations. IBM Journal of Research and Development 25(4), 272–280 (1981)
6. Giunchiglia, E., Marin, P., Narizzano, M.: sQueueBF: An Effective Preprocessor for QBFs based on Equivalence Reasoning. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing SAT 2010, Lecture Notes in Computer Science, vol. 6175, pp. 85–98. Springer Berlin Heidelberg (2010)
7. Goultiaeva, A., Van Gelder, A., Bacchus, F.: A Uniform Approach for Generating Proofs and Strategies for Both True and False QBF Formulas. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One. pp. 546–553. IJCAI’11, AAAI Press (2011)
8. Janota, M., Grigore, R., Marques-Silva, J.: On QBF Proofs and Preprocessing. CoRR abs/1310.2491 (2013)
9. Järvisalo, M., Biere, A.: Reconstructing Solutions After Blocked Clause Elimination. In: Theory and Applications of Satisfiability Testing–SAT 2010, pp. 340–345. Springer (2010)
10. Kuehlmann, A.: Dynamic Transition Relation Simplification for Bounded Property Checking. In: ICCAD. pp. 50–57. IEEE Computer Society / ACM (2004)
11. Lonsing, F., Biere, A.: Integrating Dependency Schemes in Search-based QBF Solvers. In: Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing. pp. 158–171. SAT’10, Springer-Verlag, Berlin, Heidelberg (2010)
12. Marin, P., Miller, C., Lewis, M., Becker, B.: Verification of Partial Designs Using Incremental QBF Solving. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2012. pp. 623–628 (March 2012)
13. Mishchenko, A., Chatterjee, S., Brayton, R.K., Eén, N.: Improvements to Combinational Equivalence Checking. In: ICCAD. pp. 836–843 (2006)
14. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based Certificate Extraction for QBF. In: Theory and Applications of Satisfiability Testing–SAT 2012, pp. 430–435. Springer (2012)
15. Samulowitz, H., Davies, J., Bacchus, F.: Preprocessing QBF. In: Benhamou, F. (ed.) Principles and Practice of Constraint Programming - CP 2006, Lecture Notes in Computer Science, vol. 4204, pp. 514–529. Springer Berlin Heidelberg (2006)
16. Yu, Y., Malik, S.: Validating the Result of a Quantified Boolean Formula (QBF) Solver: Theory and Practice. In: Tang, T. (ed.) ASP-DAC. pp. 1047–1051. ACM Press (2005)