Automated Benchmarking of Incremental SAT and QBF Solvers*

Uwe Egly, Florian Lonsing, and Johannes Oetsch

Vienna University of Technology,
Institute of Information Systems,
Knowledge-Based Systems Group 184/3,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{uwe,lonsing,oetsch}@kr.tuwien.ac.at

Abstract. Incremental SAT and QBF solving potentially yields improvements when sequences of related formulas are solved. An application program that employs an incremental approach to a problem is usually tailored towards some specific solver and decomposes the problem into several incremental solver calls generated directly within the application. This hinders the independent comparison of different incremental solvers, particularly when the application program is not available. To remedy this situation, we present an approach to automated benchmarking of incremental SAT and QBF solvers. Given a collection of formulas in (Q)DIMACS format generated incrementally by an application program, our approach automatically translates the formulas into instructions to import and solve a formula by an incremental SAT/QBF solver. The result of the translation is a program which replays the incremental solver calls defined by the formulas in the collection. The program can be configured to include any SAT/QBF solver which implements a minimal API similar to IPASIR, which has recently been proposed for the Incremental Library Track of the SAT Race 2015. Our approach allows to evaluate incremental solvers independently from the application program that was used to generate the formulas. We illustrate our approach by different hardware verification problems for SAT and QBF solvers.

1 Introduction

Incremental solving has contributed to the success of SAT technology and potentially yields considerable improvements in applications where sequences of related formulas are solved. Examples are bounded model checking (BMC) [4], temporal induction [11], property-directed reachability [10,8], conflict-driven clause learning combined with lookahead solving [15], and minimal unsatisfiable core computation [21,17,26,23].

The logic of quantified Boolean formulas (QBF) extends propositional logic (SAT) by explicit existential and universal quantification of variables. Problems in the complexity class PSPACE can be naturally encoded as a QBF. Incremental solving has been successfully applied to QBF encodings of problems, such as verification of partial designs [21,24], conformant planning [12], and reactive synthesis [6].

* Supported by the Austrian Science Fund (FWF) under grant S11409-N23.
The development of SAT and QBF solvers has been driven by competitive events like the SAT Competitions, SAT Races, QBFEVALs, and the QBF Galleries. These events regularly result in publicly available benchmarks submitted by the participants. Solver developers push the state of the art in SAT and QBF solving further by tuning the solvers on these benchmark sets, comparing the performance of different solvers, and inventing new solving techniques. In the past, competitive events such as the SAT Competition or the QBF Gallery focused on non-incremental SAT solving. That is, formulas are solved one by one and solvers are agnostic with regard to previously solved formulas.

Consequently, the development and evaluation of incremental solvers does not readily benefit from competitions and available benchmark collections. Testing and benchmarking incremental solvers requires to solve a sequence of related formulas. To this end, the formulas must be incrementally imported to the solver and solved by means of API calls. The API calls are typically generated by an application program—like a model checker, a formal verification tool, or a planning tool, for example—which tackles a problem by encoding it incrementally to a sequence of formulas. In order to compare different incremental solvers on that sequence of formulas, the solvers must be tightly coupled with the application program by linking them as a library.

Hence benchmarking of incremental solvers relies on the application program used to generate the sequence of formulas which, however, might not be available. Even if the application program is available, then it might have to be adapted to support different incremental solvers, each of which might come with its own API. Further, the same sequence of formulas must be generated multiple times by the application program to compare incremental solvers, which incurs run time overhead that is not related to the actual solving process. These problems affect both incremental SAT and QBF solvers.

To remedy this situation, we present an approach to automated benchmarking of incremental SAT and QBF solvers. Our approach decouples incremental SAT/QBF solving from incremental generation of formulas using an application program. The decoupling is achieved by translating a sequence of related CNFs and QBFs in prenex CNF (PCNF) into API calls of incremental solvers. To this end, an ordered sequence \( \sigma = (F_1, \ldots, F_n) \) of previously generated (or already available) formulas is syntactically analyzed. This way, the parts which are common to all formulas \( F_i \) in \( \sigma \) are identified as well as the parts in which each \( F_i \) differs from the next formula \( F_{i+1} \). Based on the common and different parts, the analysis produces instructions to incrementally import and solve the formulas in \( \sigma \) from \( F_1 \) to \( F_n \) using an incremental solver.

For CNFs, the instructions are represented as function calls in the IPASIR API, which has been proposed for the Incremental Library Track of the SAT Race 2015. For PCNFs, the instructions correspond to calls of the API of the QBF solver DepQBF, which generalizes IPASIR and supports addition and removal of quantified variables.
The result of translating a sequence $\sigma = (F_1, \ldots, F_n)$ of formulas to solver API calls is a standalone benchmarking program which replays the incremental solver calls to solve the formulas in $\sigma$ from $F_1$ to $F_n$. Any incremental SAT/QBF solver supporting the IPASIR API or its QBF extension as implemented in DepQBF can be integrated in the program by simply linking to it. The benchmarking program allows to compare different solvers on $\sigma$ without the application program that was used to generate $\sigma$.

Our approach to automated benchmarking of incremental SAT and QBF solvers underpins the goal of the Incremental Library Track of the SAT Race 2015:

*There are several SAT solvers that support incremental SAT solving, however each has its own interface, (…) That makes comparing them difficult. Our goal is to have a single interface implemented by many different solvers.*

In addition to benchmarking of incremental solvers on newly generated sequences of formulas, our approach is also applicable to already generated ones that are part of benchmark collections resulting from competitive events of the SAT/QBF communities. Our approach is independent from the application program used to incrementally generate formulas. Thereby, we make many existing benchmarks available to developers for benchmarking their incremental solvers, which enables progress in incremental solving.

In the following, we first introduce some background on incremental SAT and QBF solving. Then we present the details of our algorithm to analyze and translate sequences of CNFs and PCNFs to incremental solver API calls. We illustrate the benchmarking program which executes the API calls from a generic point of view and demonstrate its use by case studies with instances from SAT/QBF-based bounded model checking of hardware designs.

## 2 Background and Preliminaries

We consider propositional formulas in conjunctive normal form (CNF) and identify a CNF $F$ as a set of clauses. We write $\sigma = (F_1, \ldots, F_n)$ to denote a sequence $\sigma$ of clause sets $F_i$. A clause set $F_{i+1} \in \sigma$ with $1 \leq i < n$ is obtained by incrementally removing clauses from and adding clauses to the previous $F_i \in \sigma$. A sequence $\sigma$ represents the formulas that are incrementally generated and solved by an application program.

A quantified Boolean formula $\psi = P.F$ in prenex CNF (PCNF) consists of a CNF $F$ and a quantifier prefix $P$. We use lowercase Greek letters to denote QBFs and uppercase Roman letters to denote CNFs. The prefix $P = Q_1, \ldots, Q_n$ of a QBF is a sequence of pairwise disjoint quantified sets $Q_i$. A quantified set $Q$ is a set of variables with an associated quantifier $\text{quant}(Q) \in \{\exists, \forall\}$. The variables in $Q$ are existentially (universally) quantified if $\text{quant}(Q) = \exists$ ($\text{quant}(Q) = \forall$). Given a prefix $P = Q_1, \ldots, Q_n$, the index $i$ of the quantified set $Q_i$ with $1 \leq i \leq n$ is the nesting level of $Q_i$ in $P$. We write $\sigma = (\psi_1, \ldots, \psi_m)$ to denote a sequence of PCNFs $\psi_i$. The PCNF $\psi_i = P_i.F_i$ might differ from $\psi_{i+1} = P_{i+1}.F_{i+1}$ not only in the clause sets $F_i$ and $F_{i+1}$ but also in the prefixes $P_i$ and $P_{i+1}$. In addition to modifying the clause sets $F_i$ and $F_{i+1}$, quantified sets and variables may be removed from or added to $P_i$ to obtain $P_{i+1}$.

---

6 Quoted from the SAT Race 2015 website.
We assume the following syntactic restrictions of PCNFs $\psi = Q_1, \ldots, Q_n. F$. If $x \in Q_i$ for some variable $x$ and quantified set $Q_i$, then $x$ also occurs in $F$. If some variable $x$ occurs in $F$ but not in any quantified set $Q_i$, then $x$ is free in $F$. In this case, the free variable $x$ is put into an existentially quantified set $Q_0$ in the updated prefix $Q_0, Q_1, \ldots, Q_n$. This approach is similar to the handling of free variables in the QDIMACS format. All variables in $F$ are quantified and hence the PCNF $\psi$ is closed. We consider only closed PCNFs. For adjacent quantified sets $Q_i$ and $Q_{i+1}$, it holds that $\text{quant}(Q_i) \neq \text{quant}(Q_{i+1})$, that is, quantifier types alternate with each nesting level.

Our approach to automated benchmarking aims at incremental SAT and QBF solvers based on solving under assumptions, which was pioneered by MiniSAT [9,11]. Incremental solving under assumptions is compatible with conflict-driven clause learning for SAT [22,29] and QBF [14,18,30] and has been implemented in modern SAT [1,17,27] and QBF solvers [19,21,24].

Unlike earlier approaches to incremental solving [16,28], when solving a CNF under assumptions the clauses are augmented with selector variables. Selector variables are assigned as assumptions, which are temporary variable assignments made by the user via the solver API. If the value assigned to a selector variable $x$ satisfies the clauses where $x$ occurs, then these clauses are effectively removed from the CNF, including all the depending learned clauses. This way, the user controls which clauses appear in the CNF in the forthcoming incremental solver run.

The IPASIR API proposed for the Incremental Library Track of the SAT Race 2015 consists of a minimal set of functions for adding clauses to a CNF and handling assumptions. This API is sufficient for incremental SAT solving. By our approach to automated benchmarking of incremental SAT solvers we generate IPASIR calls.

For incremental QBF solving, additional API functions are needed to remove quantified sets and variables from and add them to a PCNF. For automated benchmarking of incremental QBF solvers, we decided to generate calls in the API of DepQBF, due to the lack of other publicly available incremental QBF solvers. The API of DepQBF generalizes IPASIR by functions to manipulate the quantifier prefix of PCNFs. Additionally, it allows to remove and add clauses in a stack-based way by push/pop operations [19] and, in a very recent version, supports the addition and removal of arbitrary groups of clauses [20] like the SAT solver zChaff [25]. These additional features of DepQBF come with a handling of selector variables and assumptions which is internal to the solver and hence invisible to the user.

However, for our approach to automated benchmarking, incremental QBF solvers do not need to implement the full API of DepQBF but only a minimal subset similar to IPASIR and functions to manipulate the quantifier prefix.

3 Translating Related Formulas into Incremental Solver Calls

We present the workflow to translate a given sequence $\sigma = (\psi_1, \ldots, \psi_n)$ of related PCNFs into a standalone benchmarking program which calls an integrated solver via

\[\text{http://www.qbflib.org/qdimacs.html}\]
its API to incrementally solve the formulas from $\psi_1$ up to $\psi_n$. The workflow for translating sequences of CNFs and PCNFs is the same except that for PCNFs their quantifier prefixes have to be taken into account. The translation consists of the following steps.

1. **Syntactic analysis of sequence $\sigma = (\psi_1, \ldots, \psi_n)$**: first the formulas in $\sigma$ are analyzed and the syntactic differences between each $\psi_i = P_i.F_i$ and $\psi_{i+1} = P_{i+1}.F_{i+1}$ for $1 \leq i < n$ are identified. Differences between the clause sets $F_i$ and $F_{i+1}$ are expressed in terms of clauses which are removed from or added to $F_i$ to obtain $F_{i+1}$. Differences between the prefixes $P_i$ and $P_{i+1}$ are given by quantified sets added to or removed from $P_i$ to obtain $P_{i+1}$. Additionally, some variable $x \in Q$ which appears in a quantified set $Q \in P_i$ may be removed from $Q$, and a fresh variable $y$ may be added to some quantified set $Q' \in P_i$ to obtain $P_{i+1}$. When processing sequences $\sigma = (F_1, \ldots, F_n)$ of CNFs, then comparing clause sets is sufficient and the prefix analysis is omitted.

2. **Generation of generic update instructions**: the differences between the formulas identified during syntactic analysis in the first step are expressed by generic instructions to update a formula $\psi_i = P_i.F_i$ to obtain the next formula $\psi_{i+1} = P_{i+1}.F_{i+1}$ in $\sigma$. The sequence of generic update instructions is written to a file. In order to generate generic update instructions, a clause set (i.e. CNF) $F_i$ is represented as a stack of clauses. New clauses can be added to $F_i$ by pushing them on the stack, old clauses can be removed by popping them from the stack. This way, the formulas in a sequence $\sigma = (\psi_1, \ldots, \psi_n)$ can be constructed by pop/push operations executed on the clause stack, starting from the first formula $\psi_1$. As we point out below, every sequence $\sigma$ of incrementally generated formulas can be represented by means of a clause stack and related push/pop operations, regardless of the actual application program used to generate $\sigma$. Hence our approach is applicable to incremental encodings of arbitrary problems.

The prefix $P_i$ of $\psi_i = P_i.F_i$ can be updated to obtain the prefix $P_{i+1}$ of the next formula $\psi_{i+1} = P_{i+1}.F_{i+1}$ by generic instructions to add quantified sets at specific nesting levels and to add new variables to quantified sets already present in the prefix. The generic instructions for prefix updates are inspired by the API of DepQBF, just like the incremental QBF solver calls generated finally. There are no instructions to explicitly delete quantified sets from the prefix or to delete variables from quantified sets. Instead, in our workflow an incremental QBF solver may delete a variable $x$ from a quantified set when $x$ no longer occurs in the formula, e.g., when all clauses containing $x$ have been popped from the stack. Similarly, empty quantified sets can be deleted from the prefix.

3. **Translation of generic update instructions to incremental solver calls**: the process of translating generic update instructions representing sequence $\sigma$ into API calls of incremental solvers relies on a *benchmarking program*. The benchmarking program is independent from the application program that was used to incrementally generate sequence $\sigma$. The file containing the update instructions is parsed by the benchmarking program and the instructions are translated into calls of the IPASIR API (for sequences of CNFs) or QBF solver calls (for PCNFs). For the latter, calls of DepQBF’s API are generated. However, due to its generic nature, our approach readily allows to generate API calls of other incremental QBF solvers.
The benchmarking program actually is fixed, i.e., it can be compiled once and linked with several incremental solvers. It replays the sequence of generic update instructions representing $\sigma$ in terms of incremental solver calls. This way, the performance of incremental solvers on $\sigma$ can be evaluated independently.

### 3.1 Analyzing Sequences of Formulas

We first present the algorithm to syntactically analyze sequences $\sigma = (F_1, \ldots, F_n)$ of clause sets $F_i$. This algorithm is also applied to analyze clause sets $F_i$ in sequences $\sigma = (\psi_1, \ldots, \psi_n)$ of QBFs $\psi_i = P_i, F_i$. Then we show how to analyze quantifier prefixes $P_i$ of the QBFs in $\sigma$ and generate generic update instructions to represent the formulas in $\sigma$.

#### 3.2 Analyzing CNFs

The algorithm to analyze sequences $\sigma = (F_1, \ldots, F_n)$ of clause sets relies on a stack-based representation of $F_i$. A clause $c$ which appears in some $F_i$ for the first time and is removed later at some point to obtain $F_j$ with $i < j \leq n$ is called volatile. A clause which appears in some $F_i$ for the first time and also appears in every $F_j$ with $i < j \leq n$ and hence is never deleted is called cumulative.

A stack-based representation of clause sets allows for simple deletion of clauses which have been added most recently. In contrast to deletion of clauses which were added at any point of time, with the stack-based representation it is not necessary to assign IDs to clauses when they are added to be able to remove them afterwards.

The algorithm to analyze sequence $\sigma$ identifies volatile and cumulative clauses in all clause sets in $\sigma$. Starting with $F_i$ for $i = 1$, cumulative clauses are pushed first on the stack representing the current clause set $F_i$ because they are not removed anymore after they have been added to obtain $F_i$. Volatile clauses are pushed last because they are removed at some point by a pop operation when constructing a later formula $F_j$ in $\sigma$. Volatile and cumulative clauses are formalized as follows.

**Definition 1.** Let $\sigma = (F_1, \ldots, F_n)$ be a sequence of clause sets. Within $\sigma$, a clause $c$ is cumulative in $F_i$ for $1 \leq i \leq n$, iff for all $j$ with $i \leq j \leq n$, it holds that $c \in F_j$, and either $i = 1$ or $c \notin F_{i-1}$. Clause $c$ is volatile in $F_i$ for $1 \leq i \leq n$ if $c \in F_i$ but $c \notin F_j$ for some $i < j \leq n$.

For illustration, consider the following sequence $\sigma = (F_1, \ldots, F_6)$ of clause sets $F_i$ along with their respective sets $C_i$ of cumulative clauses and sets $V_i$ of volatile clauses:

- $F_1 = \{ c_1, c_2, v_1 \}$
- $F_2 = \{ c_1, c_2, v_1, v_2 \}$
- $F_3 = \{ c_1, c_2, c_3, v_1, v_3 \}$
- $F_4 = \{ c_1, c_2, c_3, c_4 \}$
- $F_5 = \{ c_1, c_2, c_3, c_4, v_1, v_5 \}$
- $F_6 = \{ c_1, c_2, c_3, c_4, v_1 \}$

- $C_1 = \{ c_1, c_2 \}$
- $C_2 = \emptyset$
- $C_3 = \{ c_3 \}$
- $C_4 = \{ c_4 \}$
- $C_5 = \{ v_1 \}$
- $C_6 = \emptyset$

- $V_1 = \{ v_1 \}$
- $V_2 = \{ v_1, v_2 \}$
- $V_3 = \{ v_1, v_3 \}$
- $V_4 = \emptyset$
- $V_5 = \{ v_5 \}$
- $V_6 = \emptyset$
Note that the set $V_n$ of clauses which are volatile in the last clause set $F_n$ in any sequence $\sigma = (F_1, \ldots, F_n)$ of clause sets always equals the empty set. Further, the same clause can be volatile in a clause set $F_i$ and cumulative in another clause set $F_j$ with $j > i$ (like $v_1 \in V_2$ and $v_1 \in C_5$) but not the other way round.

After the sets of cumulative and volatile clauses have been identified for each $F_i$ in $\sigma = (F_1,\ldots,F_n)$, the clause sets $F_i$ in $\sigma$ can be incrementally constructed by means of operations on the clause stack which represents the respective $F_i$. The operations to modify the clause stack are

(i) adding a set $C$ of clauses permanently to a formula by $\text{add}(C)$, which can only be executed prior to any push operation,
(ii) pushing a set $C$ of clauses on the stack by $\text{push}(C)$,
(iii) popping a set $C$ of clauses from the stack by $\text{pop}(C)$, provided that $C$ has been pushed on the stack previously and was not popped in between.

The sequence $\sigma = (F_1,\ldots,F_6)$ from the example above is generated incrementally by executing the following stack operations:

\[
\begin{align*}
\text{add}(C_1) & \quad \text{push}(V_1) \\
\text{push}(C_2) & \quad \text{push}(V_2) \\
\text{push}(C_3) & \quad \text{push}(V_3) \\
\text{push}(C_4) & \quad \text{push}(V_4) \\
\text{push}(C_5) & \quad \text{push}(V_5) \\
\text{push}(C_6) & \quad \text{push}(V_6) \\
\end{align*}
\]

Note that the above schema of stack operations readily generalises to arbitrary sequences $\sigma = (F_1,\ldots,F_n)$ of clause sets. That is, in order to incrementally generate clause set $F_i$ with $1 < i \leq n$, at most two push and one pop operation is necessary, provided that the clauses have been classified in volatile or cumulative ones before.

Depending on the given sequence $\sigma$ of clause sets, in general adding and removing volatile clauses may be optimized by identifying whether $V_i \subseteq V_{i+1}$ for the sets $V_i$ and $V_{i+1}$ which are volatile in $F_i$ and $F_{i+1}$, respectively. In the example above, we have $V_1 \subseteq V_2$ and $V_1 \subseteq V_3$. Hence the volatile clause $v_1$ is added to and removed from the stack multiple times. This can be avoided in general only by allowing to remove arbitrary clauses and not just volatile ones which have been pushed most recently. However, to this end, the stack-based representation of clause sets is not appropriate and hence we have not implemented optimizations of this kind.

The algorithm for identifying cumulative and volatile clauses in a sequence of clause sets appears as Algorithm 1.

**Theorem 1.** Algorithm 1 is totally correct with respect to the precondition that $\sigma = (F_1,\ldots,F_n)$ is a sequence of sets of clauses with $n \geq 2$ and the postcondition that any $C_i$, $1 \leq i \leq n$, contains the cumulative clauses of $F_i$ in $\sigma$, and any $V_i$, $1 \leq i \leq n$, contains the volatile clauses of $F_i$ in $\sigma$.

**Proof.** Clearly, Algorithm 1 terminates on each input.

We show that the condition that any $C_j$, $1 \leq j < i$, contains the cumulative clauses of $F_j$ in the subsequence $\sigma_i = (F_i,\ldots,F_i)$ of $\sigma$, and any $V_j$, $1 \leq j < i$, contains the
volatile clauses of $F_j$ in $\sigma_i$ is an invariant of the main loop (Lines 5–19). The invariant together with $i = n$ implies the postcondition as $C_n$ always contains those clauses that are in $F_n$ but not in $F_{n-1}$ (Line 20), and $V_n$ always equals the empty set (Line 21). Likewise, the precondition implies the invariant since after Line 3, $V_1$ contains all clauses of $F_1$ that are not in $F_2$ and which are thus volatile in $F_1$ in the sequence $F_1, F_2,$ and $C_1$ contains all clauses which are in $F_1$ and $F_2$ and which are hence cumulative in $F_1$ in the sequence $F_1, F_2$.

It remains to show that if the invariant holds for some $i$, $2 \leq i \leq n-1$ at Line 6, then it holds for $i + 1$ after executing Lines 6–18. After Line 6, $V_1$ contains all the clauses that are volatile in $F_1$ in $\sigma_{i+1}$. Likewise, after Line 7, $C_i$ contains all the clauses that are in $F_i$ but not in $F_{i-1}$ and which are not volatile in $F_i$, that is, which are cumulative in $F_i$ in $\sigma_{i+1}$. Note that if a clause $c$ is volatile in some $F_j$, $j < i$, in $\sigma_i$, then $c$ is also volatile in $F_j$ in $\sigma_{i+1}$. On the other hand, if a clause is cumulative in $F_j$, it can be the case that $c$ becomes volatile in $\sigma_{i+1}$ if $c \not\in F_{i+1}$. Hence, it is possible that clauses that were previously classified as cumulative need to be reclassified.

We make use of the following claim: After Line 7, a clause $c$ is in $V_i \cap F_{i-1}$ iff, for some $j < i$, $C_j$ contains a clause $c$ that is volatile in $\sigma_{i+1}$. This claim is proven as follows: Assume that for some $j < i$, $C_j$ contains a clause $c$ that is volatile in $\sigma_{i+1}$. As the invariant holds for $i$, $c \in F_k$, for all $j \leq k \leq i$ but $c \not\in F_{i+1}$ and thus $c \in V_i$. Clearly, $c \in V_i \cap F_{i-1}$. On the other hand, assume some clause $c$ is in $V_i \cap F_{i-1}$. Clearly, $c \in V_i$ implies $c \in F_i$. Hence, as the invariant holds for $i$, $c \in C_j$, for some $j < i$, and, since $c \in V_i$, $c$ is volatile in $F_j$ in $\sigma_{i+1}$.

By virtue of the above claim, $V_i \cap F_{i-1}$ contains precisely those clauses which need to be reclassified as volatile. After Lines 9–17, for each $c \in V_i \cap F_{i-1}$, the first (and only) $C_j$ with $c \in C_j$ is found, and $c$ is removed from $C_j$ and added to all $V_i$, $j \leq i - 1$. Hence, after Lines 6–18, the invariant holds for $i + 1$. □

Note that the relevant part of the input that potentially limits scalability of Algorithm 1 is the number of variables and clauses in the formulas. The number of formulas itself is usually relatively low. By using ordered data structures, the operations on clause sets are implemented efficiently: set intersection and difference are in $O(n \cdot \log n)$, searching an element is in $O(n)$, and adding or deleting elements are in $O(1)$, where $n$ is the number of input clauses. For all experiments reported in Section 4, Algorithm 1 usually requires only a few seconds and never longer than 103 seconds (for a sequence containing 105 formulas and 26,675,128 clauses in total).

### 3.3 Analyzing PCNFs

The classification of clauses in a sequence $\sigma = (F_1, \ldots, F_n)$ into volatile and cumulative by Algorithm 1 allows to incrementally construct the clause sets in $\sigma$ in terms of operations on a clause stack. For automated benchmarking of incremental SAT solvers on sequences of clause sets, Algorithm 1 and the resulting stack operations are sufficient. For sequences $\sigma = (\psi_1, \ldots, \psi_n)$ of QBFs, additionally the differences between quantifier prefixes $P_i$ and $P_{i+1}$ of $\psi_i = P_i F_i$ and $\psi_{i+1} = P_{i+1} F_{i+1}$ must be identified. As we show below, the operations on the clause stack and the differences between
Automated Benchmarking of Incremental SAT and QBF Solvers

Input: Clause sets $F_1, F_2, \ldots, F_n$ (at least two sets are required)
Output: $C_1, \ldots, C_n$ (sets of cumulative clauses to be pushed)

$V_1, \ldots, V_n$ (sets of volatile clauses to be pushed or popped)

1. $C_i \leftarrow \emptyset$ (1 ≤ $i$ ≤ $n$);
2. $V_i \leftarrow \emptyset$ (1 ≤ $i$ ≤ $n$);
3. $V_1 \leftarrow F_1 \setminus F_2$;
4. $C_1 \leftarrow F_1 \setminus V_1$;
5. for $i \leftarrow 2$ to $n - 1$ do
6. $V_i \leftarrow F_i \setminus F_{i+1}$;
7. $C_i \leftarrow (F_i \setminus F_{i-1}) \setminus V_i$;
8. foreach $c \in V_i \cap (F_i - 1)$ do
9. // any $c$ has to be volatile in all sets $j < i$
10. for $j \leftarrow 1$ to $i - 1$ do
11. if $c \in C_j$ then
12. $C_j \leftarrow C_j \{c\}$;
13. for $k = j$ to $i - 1$ do
14. $V_k \leftarrow V_k \cup \{c\}$;
15. break;
16. end
17. end
18. end

19. $C_n \leftarrow F_n \setminus F_{n-1}$;
20. $V_n \leftarrow \emptyset$;

Algorithm 1: Identifying cumulative and volatile clauses.

prefix $P_i$ and $P_{i+1}$ finally give rise to generic update instructions which are executed by a benchmarking program to incrementally construct and solve the formulas in $\sigma$.

The analysis of quantifier prefixes is based on the following definitions.

Definition 2. Two quantified sets $Q$ and $Q'$ are matching iff $Q \cap Q' \neq \emptyset$.

Definition 3. Prefix $R$ is update-compatible to prefix $S$ iff the following conditions (i) to (iii) hold.

(i) For any quantified set of $R$, there is at most one matching quantified set in $S$.
(ii) If $P$ is a quantified set of $R$ and $Q$ is a matching quantified set in $S$, then quant($P$) = quant($Q$).
(iii) For any two quantified sets $P_1$ and $P_2$ in $S$ with matching quantified sets $Q_1$ and $Q_2$ in $R$, respectively, if the nesting level of $P_1$ is less than the nesting level $P_2$, then the nesting level of $Q_1$ is less than the nesting level of $Q_2$.

The pseudocode for generating instructions to update prefix $R$ to prefix $S$ is shown in Algorithm 2. Instructions are generated to incrementally update quantifier prefixes by adding a quantified set at a given nesting level or adding a variable to a quantified
Input: Prefix $R$ and $S$ ($R$ has to be update-compatible to $S$)
Output: Instructions to update $R$ to $S$

1. $n \leftarrow 0$
2. $m \leftarrow 0$

3. foreach quantified set $Q$ in $S$ from left to right do
4.   if $Q$ has a matching quantified set $M$ in $R$ then
5.     $m \leftarrow n +$ nesting level of $M$ in $R$;
6.     print “Add literals $Q \setminus M$ to quantified set at nesting level $m$. “;
7.   else
8.     $n \leftarrow n + 1$
9.     $m \leftarrow m + 1$
10.    print “Add quantified set $Q$ at nesting level $m$. “;
11. end
12. end

Algorithm 2: Generating update instructions for quantifier prefixes.

set at a given nesting level. For example, given the prefix $P = R,T$ where $R$ and $T$ have nesting levels 1 and 2, adding the quantified set $S$ at nesting level 2 turns $P$ into $R,S,T$.

If two quantifier prefixes $R$ and $S$ are update-compatible, then there is a sequence of instructions generated by Algorithm 2 to turn $R$ into $S$ after unused variables and empty quantified sets have been deleted.

There are no instructions to explicitly delete quantified sets or variables. Whenever a variable has no occurrences in any clause or a quantified set becomes empty, then such variables and quantified sets may be eliminated from a quantifier prefix. The incremental QBF solver for which API calls are generated by the benchmarking program is responsible to carry out simplifications of this kind.

In our experiments with automated benchmarking of incremental QBF solvers as shown below, variables or quantified sets were only added but never removed from the quantifier prefixes of the QBFs we considered. Hence simplifying the prefix was not necessary. Our solver DepQBF allows to simplify the prefix by an API call. However, to support other QBF solvers, Algorithm 2 may be adapted to also generate instructions to explicitly delete superfluous variables and quantified sets.

Algorithm 2 works as follows. For each quantified set $q$ of $S$, either it has one matching set $q'$ in $R$ (Lines 5 and 6) or it does not have a matching quantified set in $R$ (Lines 8, 9, and 10). In the former case, we need to add the atoms in $q$ to $q'$ if they are not already there (Line 6). In the latter case, we need to add the entire set $q$ to the prefix (Line 10). Adding atoms and quantified sets is always done at the right nesting level $m$. We store in $n$ the number of new quantified sets that have been added. At Line 6, when adding atoms to a matching quantified set, $m$ is the nesting level of the matching quantified set in $R$ plus the number $n$ of previously added unmatched quantified sets. At Line 10, when adding an entire quantified set, $m$ is the nesting level of the quantified set that was modified last plus one. Note that since $R$ and $S$ are update-compatible, $m$ can only monotonically increase.
3.4 The Benchmarking Program

For a sequence of $\sigma = (\psi_1, \ldots, \psi_n)$ of QBFs, Algorithms 1 and 2 produce instructions to update the clause stack, which represents the clause sets of the QBFs $\psi_i$ in $\sigma$, and the related quantifier prefixes. By executing these instructions, $\sigma$ can be incrementally constructed. In order to incrementally solve the QBFs $\psi_1$ up to $\psi_n$ in $\sigma$, the update instructions must be translated into API calls of incremental SAT or QBF solvers.

To this end, the update instructions representing a particular sequence $\sigma$ are first stored in a generic format, which is written to a file. The format is shown in Fig. 1. Instructions are interpreted as follows:

- `q <quantifier> <pnum> EOL`: add `<quantifier>` at nesting level `<pnum>`.
- `a <pnum> <pnum> EOL`: add variable `<pnum>` (first) to quantified set at nesting level `<pnum>` (second).
- `+ EOL`: add the following clauses without a preceding push operation.
- `< EOL`: push a new clause frame on the stack.
- `> EOL`: pop a clause frame from the stack.
- `<clause list>`: add clauses in `<clause_list>` to topmost frame of clause stack.

The benchmarking program is standalone and independent from the application program used to generate $\sigma$. It takes the file containing the generic update instructions as the only input and translates them into incremental calls of the API of a SAT or QBF solver. Multiple solvers may be integrated in the benchmarking program by linking them as libraries. For SAT solvers supporting the IPASIR API, stack frames pushed on the clause stack are implemented by selector variables, by which all clauses added to a particular frame are augmented. This approach may also be used for QBF solving.

Our current implementation of the benchmarking program includes DepQBF as the only incremental QBF solver, which supports push/pop operations natively via its API [19].
However, note that the push/pop instructions on the clause stack based on Algorithm 1 are not tailored towards the push/pop API of \texttt{DepQBF}. Instead, the schema of push/pop instructions illustrated by the example in Section 3.2 follows from the classification of clauses into volatile and cumulative ones by Algorithm 1.

Since we rely on generic (and hence application-independent) update instructions as an intermediate representation of sequence $\sigma$ as shown in Fig. 1 our implementation can be readily extended to support any API of SAT or QBF solvers.

4 Case Studies

In this section, we showcase our approach using different hardware verification problems for both SAT and QBF solvers. That is, for different benchmarks, we compare the performance of SAT and QBF solvers when run in incremental and non-incremental mode. Benchmark problems consist of sequences of formulas that were either generated by a model-checking tool or that were taken from existing benchmark collections. Hence, comparisons require that incremental applications are build based on such sequences and would not be possible without the methods introduced in the previous section.

All experiments were performed on a 2.53 GHz Intel Core 2 Duo processor with 4GB of main memory with OS X 10.9.5 installed.

4.1 SAT: Bounded-Model Checking for Hardware Verification

We consider problems based on the benchmarks used for the last Hardware Model Checking Competition (HWMCC 2014)\footnote{http://fmv.jku.at/hwmcc14cav/}. Competition benchmarks are given as And-Inverter Graphs (AIGs). By means of the BMC-based model checker \texttt{aigbmc}\footnote{Part of the AIGER package (http://fmv.jku.at/aiger/)} which uses \texttt{PicoSAT} as backend SAT solver, CNFs for checking AIGs can be generated incrementally. The tool \texttt{aigbmc} allows to restrict the maximal number of transitions for the considered BMC problems, and, therefore, the number of generated CNFs. For the purpose of this case study, we slightly modified \texttt{aigbmc} so that it outputs the CNFs for the internal SAT solver. Based on the CNFs, we use our tools to generate incremental solver calls and compare different SAT solvers that implement the IPASIR interface.

Note that our experiments are not intended to provide a complete comparison of all solvers on all the benchmarks. Instead, we want to illustrate our implementation on selected problems. In particular, we considered the following benchmarks: 6s280r, arbx1xs32bulp3, cuads12, cuads13, cuasq11, lmcs6dme3p3, and lmcs06bc57sp5 from the HWMCC 2014 single safety property track. Each benchmark consists of a sequence of incrementally generated CNFs, each of which correspond to a certain path length regarding the BMC problem. Formulas are unsatisfiable at first and become satisfiable as soon as a bad state is reachable. Problem 6s280r consists of 16 unsatisfiable CNFs, arbx1xs32bulp3 consists of 21 formulas (5 are satisfiable), cuads12 consists of 19 formulas (2 are satisfiable), cuads13 has 21 formulas (4 are satisfiable), cuasq11 contains...
Table 1. Different SAT solvers on hardware verification problems. If the time limit of 3600 seconds was exceeded, we report the ratio of solved CNFs.

| Benchmark | MiniSAT | PicoSAT | Lingeling | DepQBF |
|-----------|---------|---------|-----------|---------|
|           | non-inc. | inc.    | non-inc.  | inc.    | non-inc. | inc.ipa. | inc.p/p |
| 6s280r    | 10m46s  | 28m5s   | 15/16     | 15/16   | 13m0s   | 15/16    | 13/16   |
| arbixs32bugp3 | 5m8s   | 3m6s    | 13m18s    | 4m51s   | 11m15s  | 3m48s    | 17/21   |
| cuads12   | 0m37s   | 1m0s    | 2m22s     | 6m40s   | 0m35s   | 1m37s    | 16/19   |
|           | 0m36s   | 1m1s    | 2m37s     | 6m15s   | 0m37s   | 1m17s    | 16/19   |
| cuasq11   | 1m14s   | 1m14s   | 3m35s     | 7m7s    | 0m22s   | 1m11s    | 17/20   |
|           | 1m4s    | 1m32s   | 6m5s      | 20m22s  | 4m14s   | 0m58s    | 62/66   |
| lmc6dme3p3 | 4m35s  | 2m12s   | 33m15s    | 16m30s  | 16m55s  | 4m24s    | 84/105  |
| lmcs06bc57sp5 | 4m35s | 2m12s | 33m15s | 16m30s | 16m55s | 4m24s | 101/105 |

20 formulas (2 are satisfiable), lmc6dme3p3 consists of 66 formulas (6 are satisfiable), and lmcs06bc57sp5 contains 105 CNFs (2 are satisfiable).

We used the SAT solvers MiniSAT [9], PicoSAT [3], and Lingeling [5] for the considered problems. For Lingeling we implemented IPASIR. Interfaces of MiniSAT and PicoSAT are part of the IPASIR distribution. We compared runtimes for solving all formulas for a benchmark problem when solvers are used both in incremental and non-incremental mode. Also, we compared the performance of the QBF solver DepQBF version 4.0 for all problems for its non-incremental mode, the incremental mode using the solver's push/pop interface as well as an incremental mode using the IPASIR interface. The time limit for each problem and solver was set to 3600 seconds.

Table 1 summarises the results. Preparing the formulas for incremental solving by using our tools did never require more than 103 seconds. Not surprisingly, all SAT solvers clearly outperform the QBF solver DepQBF. It is however more interesting that the incremental strategy is not always better than solving the formulas independently. While DepQBF always profits from incremental solving except on cuads13, the performance of SAT solvers is more diverse. For MiniSAT and PicoSAT, on the one hand, non-incremental solving is never slower than incremental solving except on two benchmarks (arbixs32bugp3 and lmcs06bc57sp5). On the other hand, Lingeling seems to profit from incremental solving at least on four benchmarks. Especially if the overall runtime is rather small, the overhead for incremental solving seems to surpass the benefits. Regarding DepQBF, the choice between the IPASIR (column “inc.ipa.”) and its native push/pop interface (column “inc.p/p”) seems to have little impact on the performance except for arbixs32bugp3 where one additional formula is solved when using IPASIR.

4.2 QSAT: Partial Design Problems

To illustrate our approach for QBF solvers, we consider the problem of verifying partial designs. A partial design is a sequential circuit design where parts of the specification are black-boxed, i.e., a black-box in such a design corresponds to unspecified behaviour. Recent work [21,24] deals with the question whether a given safety property can be
violated regardless of the implementation of such a black-box. The authors introduce a BMC approach where this problem is translated to QBFs which are subsequently solved incrementally by means of a modified version of the QBF solver QuBE. Benchmarks (sequences of QBFs that encode partial design problems) are available from QBFLIB.

Our goal is to study the performance of DepQBF on this problem. Neither the solver used in [21,24] nor the application program used to generate sequences of QBFs are publicly available. However, our approach to automated benchmarking introduced above allows to evaluate the performance of DepQBF in incremental mode on the provided benchmarks. While Marin et al. [21] deal with preprocessing strategies for incremental solving, DepQBF does not yet apply any preprocessing. Consequently, we only consider all the benchmark instances that were used without preprocessing. Actually two incremental approaches to verification of partial designs were introduced in [21], namely forward incremental backward incremental reasoning. In a nutshell, the quantifier prefix is always extended to the right in the forward-incremental approach, while it is extended to the left in the backward incremental approach. Both strategies yield the same sequences of formulas up to renaming. The publicly available instances we used to evaluate DepQBF stem from the forward-incremental approach. Instances from the backward-incremental approach are not available.

Table 2 shows the comparison between QuBE and DepQBF. The maximal runtime of Algorithm 1 was 95 seconds. Since the modified solver used in [21,24] is not available, runtimes for QuBE in Table 2 are the ones reported in [21]. There, experiments were carried out on an AMD Opteron 252 processor running at 2.6 GHz with 4GB of main memory and a timeout of 7200 seconds. Strictly speaking, runtimes of DepQBF and QuBE are incomparable because experiments were carried out on different machines. At least, the clock rates of the machines are comparable, and experiments give a rough picture of how the performance of DepQBF compares to QuBE.

Like QuBE, DepQBF benefits from the incremental strategy on most instances. The backward-incremental strategy is clearly the dominating strategy for QuBE. A quite eye-catching observation is that forward-incremental solving, while hardly improving things for QuBE compared to the non-incremental approach, works quite well for DepQBF.

5 Conclusion and Future Work

We presented an approach to automated benchmarking of incremental SAT and QBF solvers. Our approach decouples the generation of sequences of formulas by an application program from the solving process. This is achieved by translating sequences of formulas into API calls of incremental SAT and QBF solvers executed by a benchmarking program. Several incremental solvers may be tightly integrated into the benchmarking program by linking them as libraries. We plan to make all our tools publicly available.

Competitive events such as the SAT competitions, QBFEVALs, or QBF Galleries focused on non-incremental solving in the past. Our approach to automated benchmarking allows to also evaluate incremental solvers in a competitive setting on formulas from

[10] http://www.qbflib.org
Table 2. Comparison between DepQBF and QuBE on incomplete circuit designs. Runtimes are given in seconds, \( k \) is the index of the first satisfiable formula, TO and MO refer to a timeout and memout, respectively.

| Benchmark | \( k \) | non-incremental | incremental |
|-----------|--------|-----------------|-------------|
|           | QuBE   | DepQBF          | QuBE (fwd)  | QuBE (bwd) | DepQBF |
| enc04     | 17     | 3.3 3.0         | 2.8 1.9     | 1.1        |
| enc09     | 17     | 6.9  5.3        | 6.5 4.1     | 2.5        |
| enc01     | 33     | 30.5 16.6       | 27.9 23.7   | 5.0        |
| enc03     | 33     | 32.7 16.2       | 288.9 27.6  | 27.3       |
| enc05     | 33     | 63.7 24.0       | 60.9 45.5   | 6.8        |
| enc06     | 33     | 29.2 25.5       | 27.9 23.7   | 9.9        |
| enc07     | 33     | 74.7 16.1       | 76.1 68.7   | 5.1        |
| enc08     | 33     | 108.3 16.3      | 109.6 78.8  | 5.0        |
| enc02     | 65     | 270.8 106.3     | TO 268.8    | 174.7      |
| tlc01     | 132    | 25.6 67.5       | 133.1 130.3 | 16.7       |
| tlc03     | 132    | 24.4 160.0      | 8.4 7.7     | 16.5       |
| tlc04     | 132    | 768.6 2196.1    | 1203.6 26.7 | 24.7       |
| tlc05     | 152    | 1380.4 4201.2   | 2056.7 38.1 | 33.7       |
| tlc02     | 258    | MO 42449.4      | MO 97.47    | 1907.5     |

any applications. For the first time a fair comparison of incremental solver performance on arbitrary sequences of formulas is made possible. This is particularly relevant when application programs used to generate the sequences are not available.

Additionally, automated benchmarking makes sequences of formulas which already exist in publicly available benchmark collections available for benchmarking and testing. This way, solver developers are provided with benchmarks to improve the state of the art in incremental solving.

To generate API calls of incremental SAT solvers, our approach relies on the IPASIR API which has been proposed for the Incremental Library Track of the SAT Race 2015. Hence automated benchmarking can also be used to generate competition benchmarks from arbitrary applications. Our tools generate incremental API calls for QBF solving which are inspired by the API of DepQBF. However, due to the generic nature of our approach, incremental calls of any SAT or QBF solver API can easily be generated.

We illustrated our approach to automated benchmarking of incremental SAT and QBF solvers on instances from hardware verification problems. To improve the performance of incremental QBF solving on these problems, we want to integrate incremental preprocessing into DepQBF. As shown in [21, 23], preprocessing potentially improves the performance of incremental workflows considerably.

The stack-based representation of clause sets used to generate update instructions of CNFs allows for simple deletion of most recently added (i.e., pushed) clauses and hence for a simple generic format of update instructions (Fig. 1). Deletion of arbitrary clauses can be supported by assigning IDs to clauses which, however, results in a more complex format and implementation.
Currently our approach is not feasible to analyze sequences of formulas where variables have been renamed. Although finding such renamings is a computationally hard problem in general, we plan to extend our tools in this respect.

References

1. Audemard, G., Lagniez, J.M., Simon, L.: Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction. In: Järvisalo, M., Van Gelder, A. (eds.) SAT. LNCS, vol. 7962, pp. 309–317. Springer (2013)
2. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. 25(2), 97–116 (2012)
3. Biere, A.: PicoSAT Essentials. JSAT 4(2-4), 75–97 (2008)
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, R. (ed.) TACAS. LNCS, vol. 1579, pp. 193–207. Springer (1999)
5. Biere, A.: Lingeling Essentials, A Tutorial on Design and Implementation Aspects of the the SAT Solver Lingeling. In: Le Berre, D. (ed.) Pragmatics of SAT POS Workshop. EPiC Series, vol. 27, p. 88. EasyChair (2014)
6. Bloem, R., Egly, U., Klampfl, P., Könighofer, R., Lonsing, F.: SAT-based methods for circuit synthesis. In: FMCAD. pp. 31–34. IEEE (2014)
7. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D.A. (eds.) VMCAI. LNCS, vol. 6538, pp. 70–87. Springer (2011)
8. Chockler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremental formal verification of hardware. In: Bjesse, P., Slobodová, A. (eds.) FMCAD. pp. 135–143. FMCAD Inc. (2011)
9. Eén, N., Sörensson, N.: An Extensible SAT-Solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT. LNCS, vol. 2919, pp. 502–518. Springer (2003)
10. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient Implementation of Property Directed Reachability. In: Bjesse, P., Slobodová, A. (eds.) FMCAD. pp. 125–134. FMCAD Inc. (2011)
11. Eén, N., Sörensson, N.: Temporal Induction by Incremental SAT Solving. Electr. Notes Theor. Comput. Sci. 89(4), 543–560 (2003)
12. Egly, U., Kronegger, M., Lonsing, F., Plandler, A.: Conformant Planning as a Case Study of Incremental QBF Solving. In: Aranda-Corral, G.A., Calmet, J., Martín-Mateos, F.J. (eds.) AISC. LNCS, vol. 8884, pp. 120–131. Springer (2014)
13. Giunchiglia, E., Marin, P., Narizzano, M.: QuBE7.0. JSAT 7(2-3), 83–88 (2010)
14. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. J. Artif. Intell. Res. (JAIR) 26, 371–416 (2006)
15. Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC. LNCS, vol. 7261, pp. 50–65. Springer (2011)
16. Hooker, J.N.: Solving the incremental satisfiability problem. J. Log. Program. 15(1&2), 177–186 (1993)
17. Lagniez, J.M., Biere, A.: Factoring Out Assumptions to Speed Up MUS Extraction. In: Järvisalo, M., Van Gelder, A. (eds.) SAT. LNCS, vol. 7962, pp. 276–292. Springer (2013)
18. Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In: Egly, U., Fermüller, C.G. (eds.) TABLEAUX. LNCS, vol. 2381, pp. 160–175. Springer (2002)
19. Lonsing, F., Egly, U.: Incremental QBF Solving. In: O’Sullivan, B. (ed.) CP. LNCS, vol. 8656, pp. 514–530. Springer (2014)
20. Lonsing, F., Egly, U.: Incrementally Computing Minimal Unsatisfiable Cores of QBFs via a Clause Group Solver API. In: submitted to SAT (2015)
21. Marin, P., Miller, C., Lewis, M.D.T., Becker, B.: Verification of Partial Designs using Incremental QBF Solving. In: Rosenstiel, W., Thiele, L. (eds.) DATE. pp. 623–628. IEEE (2012)
22. Marques-Silva, J., Sakallah, K.A.: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Trans. Computers 48(5), 506–521 (1999)
23. Marques-Silva, J., Lynce, I.: On Improving MUS Extraction Algorithms. In: Sakallah, K.A., Simon, L. (eds.) SAT. LNCS, vol. 6695, pp. 159–173. Springer (2011)
24. Miller, C., Marin, P., Becker, B.: Verification of partial designs using incremental QBF. AI Commun. 28(2), 283–307 (2015)
25. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC. pp. 530–535. ACM (2001)
26. Nadel, A., Ryvchin, V., Strichman, O.: Accelerated Deletion-based Extraction of Minimal Unsatisfiable Cores. JSAT 9, 27–51 (2014)
27. Nadel, A., Ryvchin, V., Strichman, O.: Ultimately Incremental SAT. In: Sinz, C., Egly, U. (eds.) SAT. LNCS, vol. 8561, pp. 206–218. Springer (2014)
28. Whittemore, J., Kim, J., Sakallah, K.A.: SATIRE: A New Incremental Satisfiability Engine. In: DAC. pp. 542–545. ACM (2001)
29. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In: ICCAD. pp. 279–285 (2001)
30. Zhang, L., Malik, S.: Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In: Hentenryck, P.V. (ed.) CP. LNCS, vol. 2470, pp. 200–215. Springer (2002)