A Dynamic QBF Preprocessing Approach for the Verification of Incomplete Designs

Christian Miller and Paolo Marin and Bernd Becker

Institute of Computer Science
University of Freiburg, Germany
millerc|paolo|becker@informatik.uni-freiburg.de

Abstract. Bounded Model Checking (BMC) is a major verification technique for finding errors in sequential circuits by unfolding the design iteratively and converting the BMC instances into Boolean satisfiability (SAT) formulas. When considering incomplete designs (i.e. those containing so-called blackboxes), we need the logic of Quantified Boolean Formulas (QBF) to obtain a precise modeling of the unknown behavior of the blackbox. The incremental nature of the BMC problem can be exploited by means of incremental preprocessing of the QBF formulas. It was shown that applying incremental preprocessing produces smaller QBF formulas, but with the price of moving computation time from the solver to the preprocessor, which can rise significantly. In this paper we show how we can take advantage of incremental preprocessing limiting this inconvenient overhead. We start applying incremental preprocessing, and switch to conventional BMC when the preprocessing effectiveness does not compensate for the time spent. We present multiple switching heuristics, and evaluate the overall performance by running several state-of-the-art QBF solvers as back-end. Comparing these results to those obtained by standard and incremental BMC procedures it turns out that dynamic preprocessing leads to the best overall performance of the whole verification process.

1 Introduction

Bounded Model Checking (BMC) is a major verification method for finding errors in sequential circuits [2,4] and one of the most successful applications of the Boolean satisfiability (SAT) problem. The BMC procedure iteratively unfolds a circuit \(k\) times, connects the negated safety property, and finally converts the BMC instance into a SAT formula. If a SAT solver finds the \(k\)-th problem instance satisfiable, a path of length \(k\) violating the property has been found.

In this paper we focus on BMC for incomplete designs, meaning that certain parts of the circuit (combined into a so-called blackbox) are not specified. The purpose is to add a layer of abstraction if a design is too large to verify in its
entirety, or allow to start the verification process earlier when certain components of the design are only partially completed. Verification of incomplete designs is emerging as larger system-on-chip (SoC) designs, containing multiple blackbox IP cores, is becoming more prevalent.

The aim of solving blackbox BMC problems is to address the question of unrealizability of a property, where finding a path of length $k$ proves that the property is violated regardless of the implementation of the blackbox. To model the unknown behavior of the blackbox outputs we make use of universally quantified Boolean variables, since this modeling allows for enough precision to prove unrealizability even if the counterexamples depend on the behavior of the blackbox [10, 16]. This yields a QBF formula, and therefore necessitates the use of a QBF solver. Note, that in contrast to other works where QBF is used for achieving a compaction of the BMC problem (e.g. [5, 11]), our need for QBF arises from the incompleteness of the design.

BMC is an intuitive example of an incremental problem, since a large portion of the problem remains the same from step $k$ to $k+1$. Indeed, incremental methods in modern SAT solvers are common, and allow them to perform significantly better on BMC problems, whereas incremental QBF solvers are not common yet, as they were introduced only in [15] for the verification of incomplete designs. Moreover, as we learned from the latest solver evaluations, the availability of multiple engines, based on heterogeneous solving techniques, allows to solve a broader set of problems [17]. For this purpose, the authors introduced incremental preprocessing in [14] to enhance the reasoning part between the BMC unfolding side and the solving phase so that any QBF solver can be used. It has been shown that incremental preprocessing is a robust procedure, although it moves part of the computation time from the solver to the preprocessor, that can be significant.

To reduce this issue we present a dynamic preprocessing policy, starting with incremental preprocessing and switching to conventional BMC if the overhead of incremental preprocessing exceeds certain thresholds. We propose two switching heuristics based on the reduction of the QBF formula the incremental preprocessor can achieve in a single unfolding step. We show that the hybrid BMC procedure can still benefit from incremental preprocessing, nevertheless improving of the overall verification process time. Furthermore, we evaluate the effectiveness of dynamic preprocessing on a range of circuit-based benchmarks, using a set of different configurations for the above heuristics, and show how the performance of the verification tool and of the back-end solver, either search-based or not, varies depending on the dynamic setting used.

The paper is structured as follows. Section 2 will first introduce the reader to QBF logic, BMC for incomplete designs, and incremental preprocessing techniques. We will then move on to the main contributions of this paper in Section 3. Section 4 will then present preliminary results we have on some circuits from www.opencores.org. Finally, Section 5 will conclude this paper.
2 Preliminaries

This section will give a brief introduction to the notation and formalisms used throughout this paper. It will start by introducing Quantified Boolean Formulas, and then move on to Bounded Model Checking of incomplete designs and incremental preprocessing techniques.

2.1 QBF Logic

QBF formulas are a generalization of pure propositional Boolean formulas where variables are either existentially or universally quantified. Most modern QBF solvers require the problem to be formatted in Quantified Conjunctive Normal Form (QCNF):

$$\psi = Q_1 x_1. Q_2 x_2 \ldots . Q_n x_n. \phi$$

(1)

where $n \in \mathbb{N}$, $Q_i \in \{\forall, \exists\}$ for all $1 \leq i \leq n$ is the prefix, and $\phi$, named matrix, is the propositional part in conjunctive normal form (CNF) over $\{x_1, \ldots, x_n\}$; this is a conjunction of clauses, which are in turn disjunctions of literals. A literal is a variable or its negation. The level of a variable $x_i$ is defined to be one plus the number of expressions $Q_j x_j. Q_{j+1} x_{j+1}$ in the prefix with $j \geq i$ and $Q_j \neq Q_{j+1}$. For our QBF problems, we expect the formula to be closed, meaning that each variable is quantified in the prefix. For more details on QBF logic, semantics and solving techniques, the interested reader is referred to [7].

2.2 BMC for Incomplete Designs

In this section we summarize the concept of verification of partial designs, which is a highly incremental application of QBF preprocessing and solving. In short, it proves the unrealizability of a safety property $P$ in an incomplete design using BMC. In other words, we prove the existence of a set of paths of length $k$ violating $P$ no matter how the blackbox, that is the unknown part of the design, is implemented.

One option to model the unknown behavior of the outputs of the blackbox is to extend Boolean logic to represent the third value ‘X’, and applying this
value to each blackbox output (this is referred to as 01X-encoding in [16]). By using this three-valued logic, we obtain a SAT problem, which can be solved in an incremental fashion. However, when the unknown value ‘X’ propagates to one of the signals which the property depends on, no information about the property can be found. In this case, those blackbox outputs need to be modeled by universally quantifying them, yielding a QBF formula.

To encode the BMC problem of incomplete designs we are naming the variables as shown in Figure 1. Here, $s_{i,j}$ denotes the $i$-th state bit in the $j$-th unfolding. The same holds for the primary inputs $x_{0,j}, \ldots, x_{n,j}$, the primary outputs $y_{0,j}, \ldots, y_{m,j}$, and the blackbox outputs $Z_{0,j}, \ldots, Z_{l,j}$. The next state variables $s_{0,j+1}, \ldots, s_{r,j+1}$ depend on the current state, the primary inputs and the blackbox outputs. The whole circuit is transformed according to [22] using additional auxiliary variables $H_j$ for each unfolding depth $j$. The initial state $I_0$ is encoded by unit clauses, setting the respective state bits to their initial value. The invariant $P_k$ can be a Boolean expression over the state variables of the $k$-th unfolding. Using this information, the quantifier prefix (and the matrix) for the unrealizability problem results in equation (2) as proposed in [16] as non-uniform prefix. For the sake of simplicity we include the variables representing the primary outputs of unfolding depth $j$ into $H_j$.

$$\exists x_{0,0} \ldots x_{n,0} \forall Z_{0,0} \ldots Z_{l,0} \exists H_0$$
$$\exists x_{0,1} \ldots x_{n,1} \forall Z_{0,1} \ldots Z_{l,1} \exists H_1$$
$$\vdots$$
$$\exists x_{0,k-1} \ldots x_{n,k-1} \forall Z_{0,k-1} \ldots Z_{l,k-1} \exists H_{k-1} \exists s_{0,k} \ldots s_{r,k}$$
$$I_0 \land T_{0,1} \land T_{1,2} \ldots \land T_{k-1,k} \land \neg P_k \tag{2}$$

2.3 Incremental Preprocessing

When producing Boolean propositional formulas in CNF, verification tools produce them starting from other formalisms, and through some translations and/or rewritings the target format is reached. Performing these steps, although highly efficient, does not usually provide an optimal result: In fact, the CNF converted problems are often much larger than their early representations, and the solving phase can be negatively affected. Thus CNF formulas are usually preprocessed before the solver’s run. During this phase, several satisfiability and/or model-preserving transformations are applied to the CNF, resulting into a shrunk formula, likely easier to solve (see, e.g. [6] for SAT or [9] for QBF).

BMC is an example of application where preprocessing can play a major role to achieve good performance. However, this clashes with the use of incremental solvers, which take new clauses and new variables after a run while keeping the

---

1 In general the property can also check the primary outputs, but for sake of convenience, we omit details in the paper.
information learned up to that point, since the preprocessor may remove variables which occur in this learned information. In [12] the authors introduced the idea of don't touch variables, to instruct the preprocessor not to simplify variables tagged in a special way, e.g. in order to prevent the elimination of the state variables. This idea was used in [15] for introducing incremental QBF solving, where a variable tagged as don't touch is an existential variable, that cannot be eliminated by Q-resolution nor by equivalence substitution.

Making use of don't touch variables, in [14] we introduced incremental preprocessing for partial design verification. This allowed to shrink the QBF formula at each unfolding depth before the actual solving process started. Doing that way, a broad range of QBF solvers could be used, and did take profit of incremental techniques. However, the preprocessor could take too long to simplify the bigger problems. Our approach for adding new information of the BMC problem to the

Algorithm 1: BMC using Incremental Preprocessing.

Input: initial state \( I \), transition relation \( TR \), property \( P \), max. unfolding depth \( K \)
Output: unrealizability of \( P \).
1. \( \Phi_{TR-pp} = \text{preprocess}(\Phi_{TR}, \{s, s'\}) \);
2. \( \Phi_{BMC} = \Phi_0^I \);
3. \( k = 0 \);
4. while \( k \neq K \) do
   5.     \( \Phi_{BMC}^k = \Phi_{BMC} \land \Phi_{k-1,k} \);
   6.     if \( \text{QBFSolve}(\Phi_{BMC}^k) == \text{true} \) then
      7.         return unrealizability proved at depth \( k \);
   8. end
   9. \( k = k + 1 \);
10. \( \Phi_{BMC} = \Phi_{BMC} \land \Phi_{TR-pp}^k \);
11. \( \Phi_{BMC} = \text{preprocess}(\Phi_{BMC}, \{s^k\}) \);
12. end
13. return Unknown;

preprocessor incrementally relies on the natural forward unfolding of BMC itself and is depicted in Algorithm 1.\(^2\) After generating and preprocessing the QCNF for the transition relation \( \Phi_{TR} \) (preprocess(\( \Phi, V \)) is a function which invokes the preprocessor over the QBF formula \( \Phi \) which variables \( V \) are tagged as don't touch), tagging the current and next state variables \( s \) and \( s' \) as don't touch, we obtain the simplified transition relation \( \Phi_{TR-pp} \) (line 1). Incremental preprocessing is done as follows: \( \Phi_{BMC} \) is a preprocessed representation of \( I_0 \land T_{0,1} \land \ldots \land T_{k-1,k} \) for any unfolding step \( k \), and is permanently stored in the preprocessor. The state variables \( s^k \) in \( \Phi_{BMC} \) are not touched, so that either the negated property \( \Phi_{k,P} \) or the next transition relation can be connected to \( \Phi_{BMC} \). \( \Phi_{BMC}^k \) adds \( \Phi_{k,P} \) to a copy

\(^2\) Since incremental preprocessing is less effective when applied to BMC in a backward manner, we restrict ourselves to forward BMC in this paper.
of $\Phi^{BMC}_k$, and thus represents the $k$-th unfolding of the BMC problem (line 5). If $\Phi^{BMC}_k$ is found satisfiable (this can be checked by any QBF solver) (line 6) the property $P$ is not realizable in the incomplete design (line 7), otherwise $k$ is incremented (line 9) and only the clauses of the next transition relation $\Phi^{k-1,k}_{TR-pp}$ are added incrementally to the preprocessor (line 10), the prefix of $\Phi^{BMC}_k$ being extended to the right (Figure 2 shows the addition of $T_{k-1,k}$ to the incremental preprocessor). The new interface variables $s_k$ are declared as don’t touch, the don’t touch variables of previous iterations get touchable again, and the preprocessor is invoked to obtain a compact representation of $\Phi^{BMC}_k$ (line 11). Starting from $I_0$, we unfold the BMC formula until the predefined maximum unfolding depth $K$ is reached. In that case, the algorithm stops returning the result Unknown, which means that the unrealizability of $P$ could not be proved within $K$ steps (line 13).

3 BMC of Incomplete Designs using Dynamic Preprocessing

When performing incremental preprocessing along all unfoldings of a BMC problem additional runtime may increase significantly. In the following, we motivate a preprocessing phase done in a dynamic fashion, that is, incremental preprocessing is performed in the initial BMC phase, and a shift to regular BMC takes place at a convenient unfolding step.

Figure 3 shows two graphs reporting solving and preprocessing times for one exemplary instance of our benchmark set ($fpu-10Xh-error01$) along its BMC unfoldings\(^3\) for various preprocessing settings. In Figure 3(a) the preprocessing time (1) and the solving time (2) for BMC using fully incremental preprocessing as described in Section 2.3, and the solving time for standard BMC without additional incremental preprocessing (3) are shown. All times are given in seconds.

\(^3\) Unrealizability can be proved at unfolding depth 27.
As expected, the solving time for standard BMC increases along the unfoldings and reaches its maximum at the last unfolding, which is the only satisfiable QBF formula of this BMC problem. This is a characteristic behavior of this kind of BMC problems. Incremental preprocessing allows to reduce the size of the QBF formulas, and in particular, it makes the satisfiable QBF formula a lot easier to solve if applied continuously. In the figure, this is reflected by the lower solving times for the incremental preprocessing mode. However, the figure also shows that preprocessing time starts to increase significantly at some point, and it is not compensated by the reduced solving time any more. Figure 3(b) shows the preprocessing time (1) and solving time (2), all in seconds, for the same exemplary benchmark when incremental preprocessing is only performed during the first eight unfoldings, thus avoiding the overhead of preprocessing the remaining unfoldings incrementally. The incrementally preprocessed formula obtained at unfolding eight is then taken as a basis for the remaining unfoldings. This is why the overall solving time can still be reduced considerably, even though the phase of incremental preprocessing is rather short. Accordingly, the overall verification time is reduced as well.

The idea of dynamic preprocessing is to stop incremental preprocessing at that point where preprocessing time starts to blow up. Since the optimal switching point is not known in advance, we now propose two heuristics to detect this point automatically during the BMC run. We introduce a metric to understand the efficiency of incremental preprocessing which relies on the quantitative reduction of the QBF formula after one unfolding. Let $\text{var}(\Phi)$ be the number of variables of a QBF formula $\Phi$ and $\text{cl}(\Phi)$ the number of clauses, respectively. The percentage reduction $\text{red}(\Phi, \Phi')$ of a QBF formula before preprocessing ($\Phi$) and after preprocessing ($\Phi'$) is defined in equation (3). The reduction of variables

\[ \text{red}(\Phi, \Phi') = \frac{\text{cl}(\Phi) - \text{cl}(\Phi')}{\text{cl}(\Phi)} \]

The authors use this metric to measure the effectiveness of incremental preprocessing and to decide when to switch to dynamic preprocessing.
and the reduction of clauses are equally weighted.

$$\text{red}(\Phi, \Phi') = \frac{\text{var}(\Phi) - \text{var}(\Phi')}{\text{var}(\Phi)} \cdot 50 + \frac{\text{cl}(\Phi) - \text{cl}(\Phi')}{\text{cl}(\Phi)} \cdot 50$$  \hspace{1cm} (3)

Since a high percentage reduction of a QBF formula may require a large pre-processing time, we additionally put this amount and the time $t$ needed to pre-process the QBF formula in relation. Thus, the final metric for the efficiency of incremental preprocessing is defined in equation (4).

$$\text{red}_t(\Phi, \Phi', t) = \frac{\text{red}(\Phi, \Phi')}{t}$$  \hspace{1cm} (4)

To put it in other words, assuming the value 10 for $\text{red}_t$, the incremental preprocessor is able to reduce the number of variables and clauses of the QBF formula by 10% every second of runtime. Our first heuristic ($\text{red}_1$) simply stops to apply incremental preprocessing from the next BMC iteration once $\text{red}_t(\Phi_{\text{BMC}}, \Phi'_{\text{BMC}}, t)$ falls below a predefined threshold. Computing $\text{red}_t$ based on $\Phi_{\text{BMC}}$ takes at each iteration also the multiple preprocessed parts of previous unfoldings into account. To avoid this, we propose a second heuristic ($\text{red}_2$) which relies on the influence of incremental preprocessing on the new part of the BMC formula, that is it measures the reduction of the new added transition relation instance $\Phi_{\text{TR-pp}}^{k-1,k}$ when connected to $\Phi_{\text{BMC}}$ and stops incremental preprocessing once $\text{red}_t(\Phi_{\text{TR-pp}}^{k-1,k}, \Phi'_{\text{TR-pp}}^{k-1,k}, t)$ falls below a given threshold.

Again, Figure 4 shows the incremental preprocessing time (1) in seconds for our example FPU design, and gives the values of $\text{red}_1$ (2) and $\text{red}_2$ (3) in %/sec. This shows how the efficiency of the incremental preprocessor decreases along the unfoldings. Experimental analysis has shown that defining a threshold of 10%/sec for $\text{red}_1$ and $\text{red}_2$, leads to suitable overall performance. For the FPU example, using this threshold would stop the incremental preprocessing phase at unfolding 7 for $\text{red}_1$ and at unfolding 2 for $\text{red}_2$, respectively.

The algorithm for using dynamic preprocessing in BMC differs from Algorithm 1 basically in the additional constraint guarding the incremental preprocessing step given a heuristic function $h \in \{\text{red}_1, \text{red}_2\}$ and a threshold $x$. We substitute line 11 with the following:

```
if $h < x$ then
    $\Phi_{\text{BMC}} = \text{preprocess}(\Phi_{\text{BMC}}, \{s^k\});$
end
```

**Requirements to the QBF Preprocessor:** The general requirements for using these approach is the possibility to add new variables, quantifier blocks, and clauses to the formula stored in the preprocessor. Furthermore, it must be able to preserve some user-defined variables (don’t touch), and dynamically switch their status into touchable. The heuristic used for switching must take into account the behavior of the built-in preprocessor: In our case, sQueueBF usually reduces the amount of clauses and variables, but makes the clauses slightly longer, and
the heuristics we proposed match that. When rather using preprocessors which apply distinct techniques these heuristics must be adapted, if not even replaced. For instance, for PreQuel [21] we would just need a different tuning of our heuristics, as it normally reduces the average amount of clauses and variables, as well. proverbox [3], which is based on the expansion of universal variables and often makes the formula larger, would probably need a completely different heuristic.

4 Experimental Results

To evaluate our dynamic preprocessing methods and switching policies, we selected some VHDL designs from www.opencores.org, and replaced parts of the circuits by blackboxes, as no partial design specifications are publicly available. We first selected an IEEE-754 compliant pipelined double precision floating point unit that supports four basic operations (+,-,*,/), multiple rounding modes and exceptions. The instances differ in their initialization settings and locations of the errors. After replacing the multiplication and division units by a blackbox and inserting an error into the circuit, proper functionality of the sign bit was falsified for addition and subtraction operations. Secondly, we used an incrementer-encoder design consisting of a configurable incrementer unit and a combinatorial logic puzzle. The instances scale with respect to both the size of the incrementer and the puzzle. With the step amount of the incrementer part blackboxed, it was falsified that the logic puzzle is solved before the incrementer reaches a certain value. Lastly, we consider a traffic light controller extended by a configurable incrementer unit. The instances differ in the complexity both of the controller and the incrementer. After blackboxing the amount added by the incrementer, we have proved unrealizability of properties expressing that one cycle of the traffic light is done before a certain value of the incrementer was reached. We use Synopsys Design Compiler Version B-2008.09 using a gate library containing only one
and two input basic logic gates and latches to synthesize a net list from the VHDL designs. Then, our BMC tool generates a QCNF representing the transition relation out of the input net list, where the information about the blackboxed parts of the design is provided. The size of these benchmarks range from about 300 gates and 26 latches (for tlc family) to 21,000 gates and 2,700 latches (for fpu family). Given the transition relation, the initial state and the property, the BMC tool produces a QDIMACS file for each BMC unfolding step. As an incremental preprocessor, we integrated sQueezeBF [9] into our BMC tool and further used various state-of-the-art QBF solvers as back-end. The evaluation was performed on an AMD Opteron 252 processor running at 2.6 GHz with 4 GB of main memory. For each BMC instance we granted a time limit of 3,600 seconds.

Table 1 shows the detailed results for our test set using the incremental version [15] of the state-of-the-art search-based QBF solver QuBE 7.2 [8]. Since variables which occur in information learned during the incremental solving phase (e.g. clauses) may be removed by the preprocessor, it is not possible to combine incremental preprocessing and incremental solving procedures in general. To this purpose, we use the non-incremental-solving version of QuBE w/o preprocessing to solve the QBF formulas obtained during the incremental preprocessing phase, and enable incremental solving features once the switching policy triggers the conventional BMC. The first column of the table names the benchmark, followed by the unfolding depth needed to prove the unrealizability of the safety property. Column 3 (std) shows the solving times for conventional BMC, i.e. using no incremental preprocessing at all, followed by BMC using fully incremental preprocessing in Column 4 (incr.pp.) split into preprocessing time (pp) and solving time (sol). Column 5, 6 and 7 give the results when applying the red1 heuristic with threshold values of 5%/sec, 10%/sec and 20%/sec, respectively, each BMC run is split into preprocessing time, solving time, and the unfolding depth when conventional BMC was triggered (sw). Columns 8, 9 and 10 are similarly structured showing the results of the red2 heuristic with threshold values of 5%/sec, 10%/sec and 20%/sec. The table sums up the preprocessing times and solving times for each BMC mode and adds a penalty of 3,600 seconds for a memout (MO) or timeout (TO). Finally, the total BMC runtime for the whole benchmark set is given. When focusing on fpu benchmarks, incremental preprocessing can reduce the solving time dramatically. However, this comes with the price of long preprocessing times. When dynamic preprocessing is enabled, this overhead is avoided. It turns out that using heuristic red1 with a threshold of 5%/sec performs best on the fpu family, that is, the policy triggers to switch from incremental preprocessing to incremental solving at unfolding 8 of the BMC problem. In contrast, best results for the inc-enc and tlc benchmarks can be achieved when incremental preprocessing is applied continuously using QuBE w/o preprocessing as back-end solver. In other words, incremental preprocessing is more beneficial to inc-enc and tlc benchmarks than incremental solving. Indeed, at every iteration the part of QBF formula of the newly added transition relation is reduced by incremental preprocessing that much, that the thresholds given by heuristic red2 never trigger from one to the other BMC procedure. Thus,
the times for red2 are similar to fully incremental preprocessing. In general, the dynamic preprocessing approach contributes to the best improvements when the transition relation is very large, thus high preprocessing times are needed. On the other hand, incremental preprocessing for smaller designs does not generally require long time: even if the unfolding depth is high, incremental preprocessing does not need to be interrupted. Taking the whole benchmark set into account, using heuristic red2 with a threshold of 5%/sec performs best, since switching to standard BMC at unfolding depth 12 is still adequate for the fpu family and no switch is performed for the remaining benchmarks. Compared to standard BMC, the dynamic preprocessing procedure can reduce the overall runtime up to 62%.

Figure 5 shows the cumulative results to solve the benchmark set for each BMC mode, again separated into preprocessing time, solving time and penalty time for memouts (MO) and timeouts (TO). Additionally to incremental QuBE, whose results were discussed in detail above, we selected QuBE with its internal preprocessor both deactivated (QuBE w/o pp) and activated (QuBE with pp). Next, we used DepQBF [13], winner of the QBFEVAL’10 competition (see www.qbflib.org), that is a search-based solver featuring prefix dependencies loosing techniques. Lastly, we selected the two portfolio-based solvers AQME [19] and Quaig [20]. Note, that the times sQueuezeBF needs to perform incremental preprocessing, and the unfolding depths of the switch to standard BMC are independent of the choice of the back-end solver. In contrast to incremental QuBE, where dynamic preprocessing can reduce the total BMC time significantly, QuBE w/o pp can take less profit, since standard BMC achieves good results anyway. Only red1-10 and red1-20 lead to a slight performance gain of a few hundred seconds. Nevertheless, these heuristics in combination with QuBE w/o pp as back-end solver perform best on the whole benchmark set, and even produce no timeouts/memouts at all. Rather, QuBE with its own preprocessor activated is not a good choice for incremental preprocessing techniques, as preprocessing an already preprocessed QBF formula may be very time consuming. In fact, dynamic preprocessing is not able to overcome this drawback and rather leads to a longer BMC runtime. Both incremental preprocessing and dynamic preprocessing have no significant influence on the BMC runtime. The same behavior can be observed for DepQBF. Dynamic preprocessing can reduce the overall verification time just a little. However, heuristic red2 produces one less timeout compared to standard BMC. The standard BMC procedure using AQME as back-end solver produces many memouts. These can be avoided almost completely using incremental preprocessing. Performing preprocessing in a dynamic fashion additionally reduces the large preprocessing time, thus, leading to a sensible reduction of the overall runtime. Quaig does not deal with preprocessed formulas at all: This can be expected, as its first step consists of an AIG-based

---

4 We also tested Quantor [1] and AIGsolve [18] to solve our BMC problems, but since these tools produced memouts on almost every instance, we do not consider these results in this paper.

5 Remember, the transition relation is also preprocessed in the standard mode.
preprocessing which transforms into AIGs the Tseitin encoded logical gates of the input QCNF. However, this encoding is destroyed by sQueezeBF.

Finally, Table 2 gives the sum of the times (preprocessing time, solving time, and penalty time for timeouts/memouts) among all solvers for the various BMC modes tested. For each of these modes, Column sum gives the total BMC time needed to solve the whole benchmark set using every QBF solver as a back-end consecutively. Again, all times are given in seconds. It clearly shows the pros and cons for standard BMC and fully incremental preprocessing. On the one hand, standard BMC needs no time for incremental preprocessing but ends up with the highest solving time and the highest timeout/memout rate among all modes. On the other hand, fully incremental preprocessing can reduce solving time to its minimum. Again, this is paid with the highest preprocessing time in this setting. Even though the results are slightly disturbed by the high penalty times for timeouts and memouts, the best compromise of incremental preprocessing and standard BMC is either using heuristic red1 with a threshold of 5%/sec or using heuristic red2 with a threshold of 5%/sec, which needs roughly 400 seconds more than the former, but is able to solve one more benchmark.

In general, BMC using dynamic preprocessing is able to reduce incremental preprocessing time still preserving the advantage of shorter solving times. This leads to an improvement of the whole verification time as well.

5 Conclusion

In this paper we presented a dynamic QBF preprocessing approach for the use in BMC for incomplete designs. Here, in the initial phase of the BMC run incremental preprocessing is applied, and heuristically interrupted to continue using conventional methods when incremental preprocessing does not pay off. For this purpose, we introduced multiple heuristics based on the efficiency of the incremental preprocessor. Experimental analysis witnesses that dynamic preprocessing prevents the incremental preprocessor to absorb high computation power yet preserving the advantage of obtaining more compact – and easier to solve – QBF formulas for each BMC instance. When compared to standard BMC and fully incremental preprocessing approaches, dynamic preprocessing leads to the best overall verification times, in particular when the transition relation of the partial design is very large.

In the future we will optimize the preprocessor, as its algorithm and data structure are not designed to manipulate such large formulas, and to adapt our switching heuristics to the improved runtimes. Next, we will study how to integrate incremental QBF solvers more tightly with the incremental preprocessing procedure.

Acknowledgments: This work has been partially funded by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, www.avacs.org)
Table 1. Results using Incremental QBE.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>k</th>
<th>std sol</th>
<th>incr. pp sol</th>
<th>red1-5</th>
<th>red1-10</th>
<th>red1-20</th>
<th>red2-5</th>
<th>red2-10</th>
<th>red2-20</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpu-10Xh-error01</td>
<td>27</td>
<td>814.01</td>
<td>609.17</td>
<td>42.32</td>
<td>12.68</td>
<td>64.29</td>
<td>7.31</td>
<td>489.31</td>
<td>7.33</td>
</tr>
<tr>
<td>fpu-10Xe-correct01</td>
<td>27</td>
<td>819.69</td>
<td>606.06</td>
<td>39.45</td>
<td>11.88</td>
<td>67.56</td>
<td>6.86</td>
<td>71.04</td>
<td>6.88</td>
</tr>
<tr>
<td>fpu-10Xh-correct02</td>
<td>27</td>
<td>785.03</td>
<td>635.77</td>
<td>35.70</td>
<td>13.13</td>
<td>55.05</td>
<td>7.65</td>
<td>64.47</td>
<td>7.81</td>
</tr>
<tr>
<td>fpu-10Xh-correct01</td>
<td>27</td>
<td>832.41</td>
<td>647.92</td>
<td>40.35</td>
<td>13.16</td>
<td>48.98</td>
<td>7.73</td>
<td>58.88</td>
<td>7.71</td>
</tr>
<tr>
<td>fpu-10Xe-correct02</td>
<td>27</td>
<td>771.41</td>
<td>638.16</td>
<td>36.00</td>
<td>13.07</td>
<td>43.38</td>
<td>7.50</td>
<td>52.79</td>
<td>7.68</td>
</tr>
<tr>
<td>fpu-10Xc-correct02</td>
<td>28</td>
<td>3232.77</td>
<td>699.37</td>
<td>40.44</td>
<td>12.30</td>
<td>58.31</td>
<td>7.31</td>
<td>75.15</td>
<td>7.27</td>
</tr>
<tr>
<td>fpu-10Xh-correct02</td>
<td>28</td>
<td>868.02</td>
<td>707.51</td>
<td>43.81</td>
<td>13.12</td>
<td>54.08</td>
<td>7.69</td>
<td>79.17</td>
<td>7.63</td>
</tr>
<tr>
<td>fpu-10Xh-correct03</td>
<td>28</td>
<td>823.22</td>
<td>655.51</td>
<td>37.23</td>
<td>12.58</td>
<td>47.04</td>
<td>7.23</td>
<td>55.68</td>
<td>7.33</td>
</tr>
<tr>
<td>inc4-enc16-01</td>
<td>17</td>
<td>2.66</td>
<td>1.18</td>
<td>2.67</td>
<td>0.83</td>
<td>2.99</td>
<td>0.52</td>
<td>2.66</td>
<td>1.02</td>
</tr>
<tr>
<td>inc4-enc16-02</td>
<td>17</td>
<td>6.33</td>
<td>2.85</td>
<td>3.40</td>
<td>0.62</td>
<td>6.59</td>
<td>0.45</td>
<td>7.00</td>
<td>2.06</td>
</tr>
<tr>
<td>inc5-enc16-01</td>
<td>33</td>
<td>11.79</td>
<td>3.82</td>
<td>11.84</td>
<td>0.97</td>
<td>14.99</td>
<td>0.46</td>
<td>16.24</td>
<td>0.31</td>
</tr>
<tr>
<td>inc5-enc16-02</td>
<td>33</td>
<td>75.45</td>
<td>4.43</td>
<td>53.01</td>
<td>0.85</td>
<td>73.36</td>
<td>0.51</td>
<td>76.94</td>
<td>0.24</td>
</tr>
<tr>
<td>inc5-enc16-03</td>
<td>33</td>
<td>197.55</td>
<td>9.90</td>
<td>63.82</td>
<td>0.73</td>
<td>163.33</td>
<td>0.47</td>
<td>166.01</td>
<td>0.28</td>
</tr>
<tr>
<td>inc5-enc32-01</td>
<td>33</td>
<td>10.83</td>
<td>3.67</td>
<td>10.79</td>
<td>0.82</td>
<td>13.84</td>
<td>0.51</td>
<td>14.83</td>
<td>0.26</td>
</tr>
<tr>
<td>inc5-enc32-02</td>
<td>33</td>
<td>66.32</td>
<td>5.16</td>
<td>40.08</td>
<td>0.87</td>
<td>64.31</td>
<td>0.41</td>
<td>62.41</td>
<td>0.19</td>
</tr>
<tr>
<td>inc5-enc32-03</td>
<td>33</td>
<td>116.10</td>
<td>6.51</td>
<td>66.19</td>
<td>0.66</td>
<td>109.94</td>
<td>0.42</td>
<td>111.80</td>
<td>0.17</td>
</tr>
<tr>
<td>inc6-enc16-01</td>
<td>65</td>
<td>1132.82</td>
<td>13.89</td>
<td>93.77</td>
<td>0.92</td>
<td>1002.70</td>
<td>0.51</td>
<td>138.55</td>
<td>0.29</td>
</tr>
<tr>
<td>inc6-enc32-03</td>
<td>65</td>
<td>32.19</td>
<td>16.64</td>
<td>9.10</td>
<td>14.16</td>
<td>8.17</td>
<td>7.00</td>
<td>6.86</td>
<td>3.48</td>
</tr>
<tr>
<td>inc-132-02</td>
<td>132</td>
<td>4.52</td>
<td>40.68</td>
<td>16.53</td>
<td>2.09</td>
<td>4.43</td>
<td>0.95</td>
<td>3.94</td>
<td>0.22</td>
</tr>
<tr>
<td>tlc-132-03</td>
<td>132</td>
<td>1014.79</td>
<td>30.61</td>
<td>254.95</td>
<td>1.95</td>
<td>892.87</td>
<td>1.08</td>
<td>809.66</td>
<td>0.50</td>
</tr>
<tr>
<td>tlc-152-01</td>
<td>152</td>
<td>1486.78</td>
<td>48.56</td>
<td>392.42</td>
<td>1.82</td>
<td>1336.76</td>
<td>0.97</td>
<td>1362.01</td>
<td>0.56</td>
</tr>
<tr>
<td>tlc-258-01</td>
<td>258</td>
<td>3600.00</td>
<td>3600.00</td>
<td>3600.00</td>
<td>3600.00</td>
<td>3600.00</td>
<td>3600.00</td>
<td>3600.00</td>
<td>3600.00</td>
</tr>
</tbody>
</table>
Fig. 5. Results using Various QBF Solvers.

<table>
<thead>
<tr>
<th>BMC mode</th>
<th>incr. preproc.</th>
<th>solving</th>
<th>timeout/memout</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>std</td>
<td>0.00</td>
<td>59765.93</td>
<td>93600.00</td>
<td>153365.93</td>
</tr>
<tr>
<td>incr.pp.</td>
<td>26070.19</td>
<td>39856.49</td>
<td>68400.00</td>
<td>134326.68</td>
</tr>
<tr>
<td>red1-5</td>
<td>665.94</td>
<td>49563.70</td>
<td>64800.00</td>
<td>115029.64</td>
</tr>
<tr>
<td>red1-10</td>
<td>380.48</td>
<td>51637.93</td>
<td>64800.00</td>
<td>116093.18</td>
</tr>
<tr>
<td>red1-20</td>
<td>342.92</td>
<td>50950.26</td>
<td>64800.00</td>
<td>116818.41</td>
</tr>
<tr>
<td>red2-5</td>
<td>3584.60</td>
<td>50627.15</td>
<td>61200.00</td>
<td>115411.75</td>
</tr>
<tr>
<td>red2-10</td>
<td>1143.00</td>
<td>54712.25</td>
<td>64800.00</td>
<td>120655.25</td>
</tr>
<tr>
<td>red2-20</td>
<td>1137.11</td>
<td>54151.83</td>
<td>64800.00</td>
<td>120088.94</td>
</tr>
</tbody>
</table>

Table 2. Cumulated Results for All Solvers.
References