Verification of Partial Designs Using Incremental QBF Solving

Paolo Marin and Christian Miller and Matthew Lewis and Bernd Becker
Albert-Ludwigs Universität Freiburg, Germany
Email: {paolo,millerc,becker}@informatik.uni-freiburg.de

***Abstract*—SAT solving is an indispensable core component of numerous formal verification tools and has found widespread use in industry, in particular when using it in an incremental fashion, e.g. in Bounded Model Checking (BMC). On the other hand, there are applications, in particular in the area of partial design verification, where SAT formulas are not expressive enough and a description via Quantified Boolean Formulas (QBF) is much more adequate.**

In this paper we introduce incremental QBF solving and thereby make it usable as a core component of BMC. To do so, we realized an incremental version of the state-of-the-art QBF solver QuBE, allowing for the reuse of learnt information e.g. in the form of conflict clauses and solution cubes. As an application we consider BMC for partial designs (i.e. designs containing so-called blackboxes) and thereby disprove realizability, that is, we prove that an unsafe state is reachable no matter how the blackboxes are implemented. In our experimental analysis, we compare different incremental approaches implemented in our BMC tool. BMC with incremental QBF turns out to be feasible for designs with more than 21,000 gates and 2,700 latches. Significant performance gains over non incremental QBF based BMC can be obtained on many benchmark circuits, in particular when using the so-called backward-incremental approach combined with incremental preprocessing.

I. Introduction

SAT solving is a mature technology, the increase of its power over the last decade has been astounding. It is now an indispensable core component in many hardware and software formal verification tools. Moreover, it has found widespread application for verification in industry, in particular when using it in an incremental fashion [1], like in Bounded Model Checking (BMC) [2], [3], where most of the formula remains the same for consecutive instances along the unrolling. This way, the problems can be tackled more efficiently thanks to the reuse of much of the information learnt step by step (i.e. conflict clauses, literal activities) [4], [5]. On the other hand, for some applications SAT formulas are not expressive enough, for instance in the area of partial design verification, and a representation using Quantified Boolean Formulas (QBF) is required [6], [7].

In this paper we introduce incremental QBF solving, where we take into consideration both the propositional part and the dependencies between old and new variables. We build upon many of the incremental techniques used in SAT solving, discussing the requirements that allow a solver to keep the information it learnt from one call to the next one, and how to modify a search-based QBF solver to work incrementally.

978-3-9810801-8-6/DATE12/©2012 EDAA

We have realized an incremental QBF solver built upon the state-of-the-art QBF solver QuBE [8]. Moreover, we present the first application of incremental QBF solving as the verification of *incomplete* designs, where certain parts of the circuit (combined into a so-called blackbox) are not specified. The interest in verifying incomplete designs is emerging as larger system-on-chip (SoC) designs, that contain multiple blackbox IP cores, have become more prevalent. Blackboxes can also be used to verify designs which are too large to verify in their entirety by adding a layer of abstraction. Lastly, they allow us to start the verification process earlier in the design stages of a chip when certain components are only partially completed.

In BMC for incomplete designs we look for an answer to the question of *unrealizability*, that is, if there exists a path of length k violating the property regardless of the implementation of the blackbox. If it exists, the property is unrealizable. The unknown behavior of the blackbox outputs could be modeled using 01X-logic [9], yielding a SAT formula, but it may be too coarse when the counterexample depends on the blackbox's behavior. In that case we can rely on QBF-logic and quantify the blackbox's outputs universally [6], [7]. For the purpose of obtaining a more compact representation of each unfolding step, BMC was also performed using Quantified Boolean Formula in [10], [11], but at that time no QBF solver supported incremental solving, nor was there any theoretical groundwork proposed. To solve our partial design verification problems using incremental QBF, we introduce two approaches (namely. backward and forward unfolding), and highlight the advantages and disadvantages of both proposed methods, in particular regarding the possibility of keeping learnt information. We also show on a range of incomplete design benchmarks (designs having more than 21,000 gates and 2,700 latches) that QBF BMC is feasible using incremental QBF, and significant performance gains can be obtained, especially when using the so-called backward-incremental approach in combination with incremental preprocessing.

The paper is structured as follows. Section II first introduces the reader to QBF. After this, Section III presents the incremental QBF problem, and how to modify a search-based solver to take it. We then describe our partial design verification problems, and explain how to use incremental QBF to solve them in Section IV. Section V presents some preliminary results we have on circuits available from Opencores [12]. Finally, Section VI concludes this paper.

II. Formal Preliminaries

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). This format consists of a prefix and a matrix. The prefix of the formula defines how each variable is quantified, and the interdependencies between all the variables (represented by the order in which they are quantified). Each quantifier alternation defines the next quantification level. The matrix consists of a conjunction of clauses, with each clause consisting of the inclusive disjunction of literals. Each literal represents the occurrence of a Boolean variable in a clause. A literal can take on the positive or negative form of the variable it represents. For our QBF problems, we expect the formula to be closed, meaning that each variable is quantified in the prefix. In search-based QBF solvers it is common to make use of additional clauses, determined and learned into a separate CNF formula Ψ while backtracking from conflicts, to avoid unsatisfiable parts of the search space, and likewise to determine and learn cubes (sometimes referred to as terms) during the solution analysis to truncate the solution space. Each cube consists of a conjunction of literals, and cubes are stored in a Disjunctive Normal Form (DNF) formula Θ, which is treated as a disjunction together with the CNF matrix. Given an input formula Φ, these QBF solvers solve its logically equivalent form $\Psi \wedge \Phi \vee \Theta$, also called *Extended QBF* (EQBF) in [13]. For more details on QBF logic, semantics and solving techniques, the interested reader is referred to [14].

III. Incremental QBF Solving

In this section we generalize to QBF the idea of incremental solving as it was defined in [4]. Given at step 0 an arbitrary prenex CNF formula Φ_0 with prefix $Q_1 X_1 \dots Q_n X_n$ and matrix ϕ_0, the formula Φ_i at step i is the merge of the prefix at step $i-1$ and the prefix at step i, and the matrix is obtained by conjuncting the new clauses ϕ_i^+ to ϕ_{i-1} and/or removing old clauses ϕ_i^- from ϕ_{i-1}. A critical point of incremental solving such problems lies in the information depending on ϕ_i^- held from the solver at step i. Indeed, modern QBF solvers use learning techniques that improve their performance, and incremental solving takes advantage of that. What is then necessary, is to be able, at the beginning of step i, to delete all the learnt clauses and cubes that depend on ϕ_i^-. In QBF we also have to take care of the learnt cubes from an additional point of view: In general, at every step, we must delete all the cubes learnt previously. This however does not always apply: It can be proved that old cubes are still sound if:

- the new variables do not depend on the old ones
- in ϕ_i^+ there are no clauses having literals that depend on universal variables from previous steps.

Remember that in search-based QBF solving, to decide heuristically the value to give an unassigned variable, the prefix order matters: For instance, it is not sound to decide a value for a variable, if some variables in the quantifier blocks to its left are still unassigned. Now consider a cube: It can only be activated (made unit or empty) by a unit or a decision universal literal. Since a universal literal can only be implied by a cube, it turns out that the old cubes will not be activated until a decision over a universal variable in the old universal quantifier blocks (which depends on the new variables) is taken. Hence, all the new variables must have been assigned a value without producing any conflict, thus satisfying ϕ_i^+, before the old cubes are activated.

Our starting point for extending a QBF solver by incremental means is the state-of-the-art search-based solver QuBE7.2 [8], [15]. The first new feature QuBE7.2 needed to support was to solve QBF formulas modulo assumptions. Assumptions act as decision literals forced from outside the solver, with the difference being that they are assigned at decision level 0. As proposed in [4] for incremental SAT, we use an additional existential variable called "assumption" to activate and deactivate clauses and solution cubes depending on ϕ_i^- before each new iteration. To keep the assumptions into the conflict clauses (and solution cubes) produced by the conflict (and solution) analysis procedure, this is modified not to discard from the simplifications the variables assigned at level 0. Furthermore, in QBF we may face *long-distance* resolutions performed by the conflict (and solution) analysis (also called *recursive resolutions*) [13]. Binding the assumption variable to the outermost existential quantification level of the prefix does not affect the above mentioned mechanism.

The solver can basically take advantage of solving the problems incrementally in two ways: (*i*) if every incremental matrix has always the same structure, the new variables can inherit the activity score from those of their counterparts of the previous iteration, and (*ii*) the search space can be pruned using the conflict clauses learnt previously. Solution cubes learnt previously can be used only if the conditions above hold.

IV. Using Incremental QBF for BMC of Incomplete Designs

In this section we present an application for incremental QBF solving, namely proving unrealizability of a safety property P in an *incomplete* design (those containing so-called blackboxes) using BMC. In other words, we prove the existence of a path of length k violating P no matter how the blackboxes are implemented. One option to model the unknown output of the blackboxes is to extend Boolean logic to represent the third value 'X' using an additional Boolean variable, and applying this value to each blackbox output (this is referred to as 01X-encoding [16]). By using this three valued logic, we obtain a SAT problem, which can be solved in an incremental fashion. However, if it happens that the unknown value 'X' propagates to one of the circuits outputs 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.

For the encoding of the BMC problem of incomplete designs we are naming the variables as shown in Fig. 1. Here,

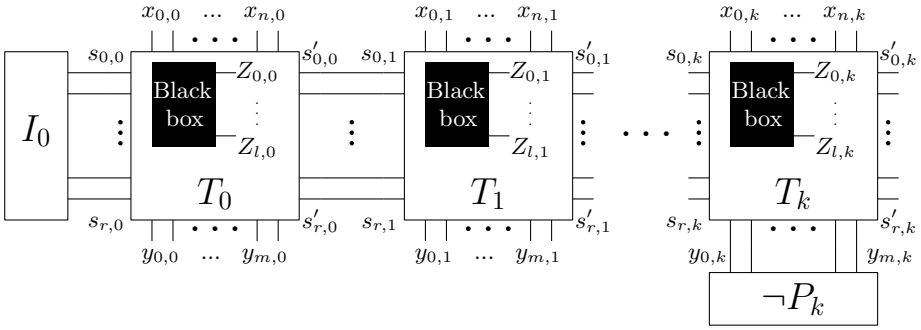

Fig. 1. Encoding of the BMC problem.

$s_{i,j}$ denotes the ith state bit in the jth 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 $s'_{0,j}, \ldots, s'_{r,j}$ which depend on the current state, the primary inputs and the blackbox outputs are then connected to the current state bits of the next state $j+1$. The whole circuit is transformed according to [17] 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 bit to its initial value. The invariant P_k can be a Boolean expression over the primary outputs and the state variables of the k-th unfolding. Using this information, the quantifier prefix (and the matrix) for the unrealizability problem is:

$$\exists x_{0,0} \ldots x_{n,0} s_{0,0} \ldots s_{r,0} \forall Z_{0,0} \ldots Z_{l,0} \exists H_0 s'_{0,0} \ldots s'_{r,0}$$
$$\vdots$$
$$\exists x_{0,k} \ldots x_{n,k} s_{0,k} \ldots s_{r,k} \forall Z_{0,k} \ldots Z_{l,k} \exists H_k s'_{0,k} \ldots s'_{r,k}$$
$$I_0 \wedge T_0 \wedge \ldots \wedge T_k \wedge \neg P_k$$

as proposed in [7] as non-uniform prefix. For the sake of simplicity we include the variables representing the primary outputs of unfolding depth j into H_j.

We now show how we have extended [4] to QBF in the context of the verification of incomplete designs. This could be done in many ways, as we can play not only with adding and removing clauses and variables, but also adding or removing quantifier blocks. We introduce two incremental strategies where the prefix is augmented at its extremes.

A. Forward-Incremental BMC for Incomplete Designs

Our first approach for adding new information to the BMC problem incrementally relies on the nature of the BMC problem itself. We start at depth 0 with $I_0 \wedge T_0 \wedge \neg P_0$ to check whether the property is violated in the initial state (we add T_0 to have access to the primary outputs at depth 0). Thus, the initial prefix for the QBF solver contains three quantifier blocks and is denoted in Fig. 2(a). The matrix ϕ_0 contains the clauses resulting from the Tseitin transformation of all gates in T_0 and $\neg P_0$, and the unit clauses for the initial state I_0. For the next unfolding depth k, the prefix of the previous depth is then extended to the right, adding two new quantifier blocks, and merging the inner existential quantifier block at depth $k-1$ and the outer existential quantifier block at depth k. The matrix is augmented by the set ϕ_k^+ containing only the clauses representing the transition relation T_k, and those representing the negated property $\neg P_k$ (Fig. 2(a)).

By doing incremental solving this way, we can keep much of the information which was learnt during the solving process of the previous depth $I_0 \wedge T_0 \wedge \ldots \wedge T_{k-1} \wedge \neg P_{k-1}$. However, since keeping the clauses for $\neg P_{k-1}$ would lead to unsatisfiability of all future unfolding depths, these clauses must be inserted in ϕ_k^- and deleted, as well as all learnt information resulting from these clauses. This is done by using the assumptions as explained in Section III. In addition to this, we must delete all solution cubes, as they will no longer be valid if we add quantifier levels to the inside of the prefix. In fact, since new variables depend on the old ones, there may exist old cubes which are triggered when new variables still have to be assigned.

B. Backward-Incremental BMC for Incomplete Designs

In the forward approach it is not sound to keep learnt solution cubes in the incremental BMC process. To overcome this problem, we now present a backward-incremental procedure in which the prefix is extended to the left at each unfolding depth. In this approach we start at depth 0 again with $I_0 \wedge T_0 \wedge \neg P_0$, checking for an initial violation of the property. For all future unfolding depths k we extend the matrix (and the prefix) backwards. This means, at step k, the clauses for the initial state $\phi_k^- = I_{k-1}$ are disabled, and the set ϕ_k^+ including the transition relation T_k, as well as the new initial state I_k, is added (Fig. 2(b)).

Actually, with backward-incremental BMC the same problem instances (except for variable renaming) are generated as for the forward-incremental procedure, as depicted in Fig. 3. But using this approach we can now keep, and reuse, all learnt information that does not depend on the old initial state, as we now fulfil the conditions stated in Section III, allowing for also the solution cubes to be kept.

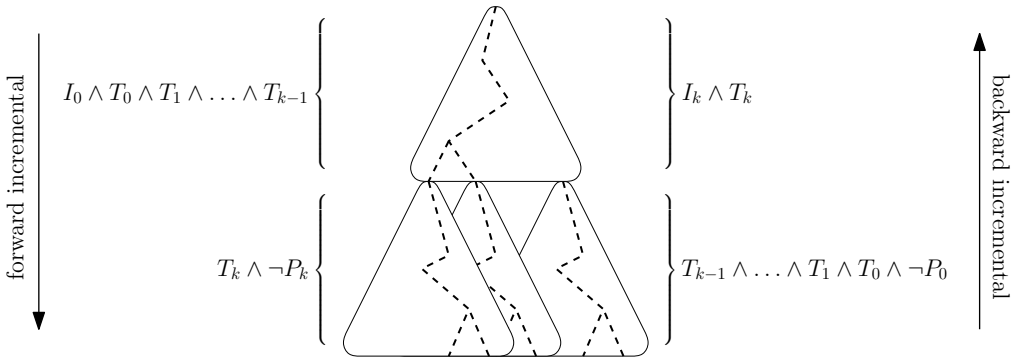

Fig. 3. Extension of the search space for the incremental approaches.

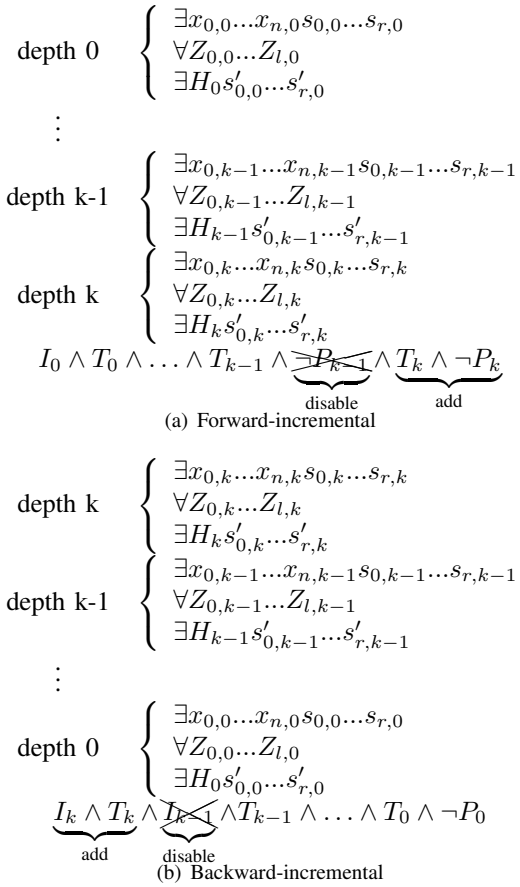

Fig. 2. QBF encoding of the BMC problem using the incremental approaches.

C. QBF Preprocessor Modifications

We extended the incremental BMC preprocessing methods for SAT presented in [18] to the QBF domain by modifying sQueezeBF [19], the preprocessor built into QUBE7.2, to preprocess the QBF formulas incrementally. In essence, we adopted the idea of *don't touch literals* for preprocessing the transition relation: The idea is to take the list of the state and next state variables, and ensure these not to be eliminated from the output formula. By incorporating this idea we obtain smaller sets of clauses and variables which have to be added at every step. However, not *touching* a (next) state variable means that we can neither eliminate that variable by Q-resolution, nor by equivalence reasoning [19]. This can sometimes restrict the effectiveness of the preprocessor. The preprocessing step is performed independently before the incremental search starts.

V. Experimental Results

To test our novel forwards and backwards-incremental QBF approaches for verifying incomplete designs, we implemented an extension to our blackbox BMC tool [7] that allows us to work incrementally. As a test-bed, we selected some VHDL designs from Opencores [12], and replaced parts of the circuits by blackboxes. We did this because there is currently no blackbox design problems (or increment QBF problems) publicly available. We first used 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 inserted error locations. After replacing the multiplication and division units by blackboxes 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 proven unrealizability of properties expressing that one cycle of the traffic light is done before a certain value of the incrementer was reached. The size of these benchmarks range from about 300 gates and 26 latches to 21,000 gates and 2,700 latches. The VHDL designs were compiled with Synopsys Design Compiler Version B-2008.09 using a minimized gate library containing only one and two input basic logic gates and latches. All the QBF formulas generated can be freely downloaded from QBFLIB [20]. The experiments were performed on an AMD Opteron 252

TABLE I
INCREMENTAL QBF RESULTS ON INCOMPLETE CIRCUIT DESIGNS

(a) Results without Preprocessing

benchmark	k	non-incremental		forward-incremental		backward-incremental	
		time	#dec	time	#dec	time	#dec
inc4-enc16-01	17	3.13	20,732	2.75	20,286	**1.85**	11,892
inc4-enc16-02	17	6.88	93,938	6.49	104,692	**4.10**	72,660
inc5-enc16-01	33	30.48	94,888	27.93	93,103	**23.72**	65,739
inc5-enc16-02	33	32.65	100,324	288.88	676,013	**27.55**	71,191
inc5-enc16-03	33	63.72	384,787	60.86	384,712	**45.45**	284,750
inc5-enc32-01	33	29.16	94,888	27.89	93,103	**23.65**	66,491
inc5-enc32-02	33	74.73	133,460	76.10	179,187	**68.72**	112,372
inc5-enc32-03	33	108.29	372,391	109.59	362,929	**78.78**	215,530
inc6-enc16-01	65	270.80	421,035	TIMEOUT		**268.81**	347963
tlc-132-01	132	**25.64**	1,553	133.12	1,553	130.34	1,553
tlc-132-02	132	24.35	1,553	8.36	1,553	**7.71**	1,553
tlc-132-03	132	768.58	747,764	1,203.63	759,961	**26.65**	17,445
tlc-152-01	152	1,330.43	1,117,627	2,056.70	1,081,608	**38.09**	21,666
tlc-258-01	258	MEMOUT		MEMOUT		**97.47**	93,506

(b) Results with Preprocessing

benchmark	k	non-incremental		forward-incremental		backward-incremental	
		time	#dec	time	#dec	time	#dec
fpu-10Xh-error01	27	724.88	2,339	625.68	19,594	**606.31**	24,586
fpu-10Xh-error02	27	815.30	2,339	621.53	19,594	**602.50**	24,586
fpu-10Xe-correct01	27	593.80	149	606.19	13,740	**576.71**	14,272
fpu-10Xh-correct01	27	821.22	2,222	624.94	18,440	**604.67**	23,432
fpu-10Xh-correct03	27	722.44	2,143	590.02	3,182	**563.91**	3,455
fpu-10Xe-correct02	28	**800.34**	131	2,383.94	17,340	1,994.26	17,894
fpu-10Xh-correct02	28	1,009.46	2,510	676.01	19,704	**661.01**	25,207
fpu-10Xh-correct04	28	709.45	212	629.90	3,193	**609.26**	3,466
inc4-enc16-01	17	7.02	12,748	2.15	13,525	**1.64**	12,867
inc4-enc16-02	17	13.34	73,852	5.11	88,128	**4.93**	83,768
inc5-enc16-01	33	29.43	19,857	8.95	22,296	**8.61**	10,946
inc5-enc16-02	33	70.04	86,753	**56.68**	112349	57.24	98,664
inc5-enc16-03	33	**121.83**	394924	142.36	502423	151.39	546,768
inc5-enc32-01	33	29.99	19,857	8.85	22,296	**8.54**	20,946
inc5-enc32-02	33	66.21	80,398	52.90	89,777	**50.95**	113,038
inc5-enc32-03	33	111.35	260,433	94.93	250,004	**84.38**	227,278
inc6-enc16-01	65	201.19	56,098	TIMEOUT		**101.11**	63,858
tlc-132-01	132	111.62	248	25.32	249	**25.29**	249
tlc-132-02	132	164.54	1,536	3.71	1,553	**3.68**	1,553
tlc-132-03	132	520.41	655,991	714.72	545,511	**13.72**	14,129
tlc-152-01	152	753.16	995,600	1,325.41	813,496	**16.73**	17,265
tlc-258-01	258	MEMOUT		MEMOUT		**44.34**	103,441

processor running at 2.6 GHz with 4 GB of main memory and a timeout of 7,200 seconds.

Table I(a) shows our results on the benchmarks described above without preprocessing. The `fpu*` family was excluded as none of the configurations were able to solve any of them. For each benchmark, the depth k at which unrealizability could be proven is stated. This is followed by the results for the non-incremental, forward-incremental, and backward-incremental solvers. For each mode the solve time in seconds and the number of decisions over all unfoldings (0 to k) are provided. It can be seen that the forward-incremental approach does not provide a good improvement from the non-incremental case. Rather, backward-incremental QBF solving seems to provide the best performance gain (bold values): this is obtained by a reduction in time, and a constantly smaller number of decisions needed to prove unrealizability, also when compared with *forward-incremental*. Basically, the learnt information helps the search, preventing certain parts of the search space from being searched multiple times as would be needed in the non-incremental case. Moreover, this happens even more frequently in the forwards setting, but the greater amount of decisions made by the solver shows that those previously learnt clauses do not always drive the search in the best direction. As explained in Section IV-B, the solution cubes learnt by the solver in the backward-incremental approach can be very general, and similar effects benefit conflict clauses as well.

Table I(b) shows the results when performing preprocessing during the BMC process, and is organized as Table I(a). It is evident that the backward-incremental approach with preprocessing is the winning strategy for solving the most benchmarks (bold values). Note, in many cases, the number of decisions taken by the solver is lower in the non-incremental approach. This is due to the fact that the non-incremental method allows for a more effective preprocessing, allowing the (next) state variables (*don't touch*) to be eliminated. However, because of the complete preprocessing phase repeated at every

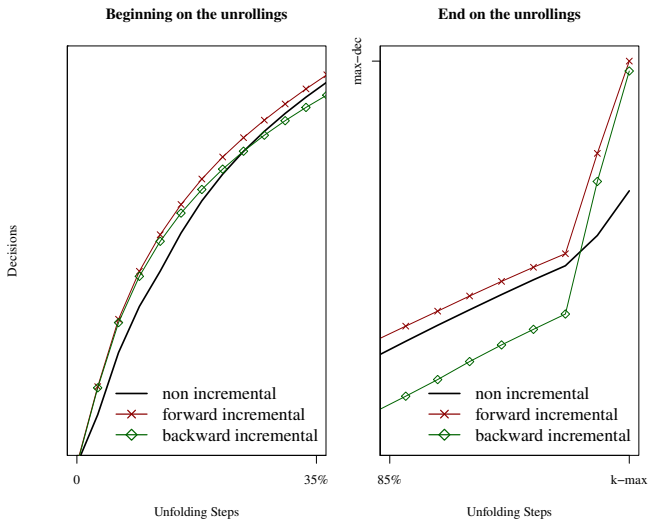

Fig. 4. Number of branches explored along the verification process.

unrolling step overall BMC times are higher. In Fig. 4 we display the progress of the search computed by the solver (logarithmic scale) at the beginning (left) and at the end (right) of the BMC procedure, in terms of decisions taken, when using preprocessing. We only considered the benchmarks that all the three (incremental and non) versions could solve.

Looking at the left plot, we notice that thanks to the (unconstrained) preprocessor, the non incremental solver has to make less decisions. Indeed, when using *don't touch* variables, the preprocessor is less effective. This also makes the incremental solving procedure explore a larger space in the last part of the process (right plot). We do not show the plot for the non-preprocessed case as there are no intersections between the lines, nor other unexpected comportments. Overall, the performance gains of our verification tool are best when using backward-incremental and preprocessing methods.

VI. Conclusion and Future Work

This paper introduced incremental QBF solving, and described how a QBF solver can be made to work incrementally. We demonstrated the usefulness of incremental QBF solving in an exemplary application area, Bounded Model Checking of incomplete designs. We discussed two possible ways to solve the problem incrementally using modern search based QBF solvers. For both the forward and backward-incremental approaches, we discussed the advantages and limitations of each idea. Regarding preprocessing, we highlighted some of the problems associated with preprocessing and incremental QBF solving and presented possible solutions. Finally, results were presented showing that the backward-incremental approach seems to be the most promising.

In the future, we plan to continue the development of our tool, and incorporate circuit based optimizations such as cone-of-influence reductions instead of always using the transition relation in its entirety. Furthermore, we want to extend our incremental BMC tool for partial designs to the uniform quantifier prefix: This should allow us to verify larger and more complex designs in the future, and allow to take in input formulas coming from other domains as well, for example for QBF based BMC of complete designs as proposed in [10], [11].

Acknowledgments

The authors would like to thank STAR-lab of the University of Genova, Italy, for fruitfully cooperating on QuBE. Additionally, this work was partly supported 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). See www.avacs.org for more information.

References

[1] J. N. Hooker, "Solving the incremental satisfiability problem," *Journal of Logic Programming.*, pp. 177–186, 1993.
[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 1999, pp. 193–207.
[3] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design.*, pp. 7–34, 2001.
[4] N. Een and N. Sörensson, "Temporal Induction by Incremental SAT Solving," *Electr. Notes Theor. Comp. Sci.*, vol. 89, 2003.
[5] O. Shtrichman, "Pruning Techniques for the SAT-Based Bounded Model Checking Problem," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods.*, 2001, pp. 58–70.
[6] M. Herbstritt and B. Becker, "On Combining 01X-Logic and QBF," in *Proceedings of 11th International Conference on Computer Aided Systems Theory (EuroCAST)*. Springer Verlag, 2007, pp. 531–538.
[7] C. Miller, S. Kupferschmid, M. Lewis, and B. Becker, "Encoding Techniques, Craig Interpolants and Bounded Model Checking for Incomplete Designs," in *Theory and Applications of Satisfiability Testing*, 2010.
[8] Giunchiglia, E., and Marin, P., and Narizzano, M., "QuBE7.0, System Description," *Journal of Satisfiability.*, pp. 83–88, 2010.
[9] M. Herbstritt, B. Becker, and C. Scholl, "Advanced SAT-Techniques for Bounded Model Checking of Blackbox Designs," in *Proc. Workshop on Microprocessor Test and Verification*, 2006, pp. 37–44.
[10] N. Dershowitz, Z. Hanna, and J. Katz, "Bounded Model Checking with QBF," in *Theory and Applications of Satisfiability Testing*, 2005, pp. 408–414.
[11] T. Jussila and A. Biere, "Compressing BMC Encodings with QBF," *Electr. Notes Theor. Comput. Sci.*, vol. 174, no. 3, pp. 45–56, 2007.
[12] "OpenCores," http://opencores.org/.
[13] E. Giunchiglia, M. Narizzano, and A. Tacchella, "Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas," *Journal of Artificial Intelligence Research.*, vol. 26, pp. 371–416, 2006.
[14] E. Giunchiglia, P. Marin, and M. Narizzano, *Reasoning with Quantified Boolean Formulas*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185, ch. 24, pp. 761–780.
[15] P. Marin, E. Giunchiglia, and M. Narizzano, "Conflict and Solution Driven Constraint Learning in QBF," in *Doctoral Program of Constraint Programming Conference*, 2010.
[16] C. Scholl and B. Becker, "Checking Equivalence for Partial Implementations," in *Design Automation Conference*, 2000, pp. 238–243.
[17] G. Tseitin, "On the Complexity of Proofs in Propositional Logics," *Seminars in Mathematics*, 1970.
[18] S. Kupferschmid, M. Lewis, T. Schubert, and B. Becker, "Incremental Preprocessing Methods for use in BMC," in *Int'l Workshop on Hardware Verification*, 2010.
[19] E. Giunchiglia, P. Marin, and M. Narizzano, "sQueezeBF: An Effective Preprocessor for QBFs Based on Equivalence Reasoning," in *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2010.
[20] "Quantified Boolean Formulas satisfiability library," www.qbflib.org.