Bounded Model Checking with QBF

N. Dershowitz, Z. Hanna and J. Katz

Abstract. Current algorithms for bounded model checking (BMC) use SAT methods for checking satisfiability of Boolean formulas. These BMC methods suffer from a potential memory explosion problem. Methods based on the validity of Quantified Boolean Formulas (QBF) allow an exponentially more succinct representation of the checked formulas, but have not been widely used, because of the lack of an efficient decision procedure for QBF. We evaluate the usage of QBF in BMC, using general-purpose SAT and QBF solvers. We also present a special-purpose decision procedure for QBF used in BMC, and compare our technique with the methods using general-purpose SAT and QBF solvers on real-life industrial benchmarks. Our procedure performs much better for BMC than the general-purpose QBF solvers, without incurring the space overhead of propositional SAT.

1 Introduction

Model checking is a technique for the verification of the correctness of a finite-state system with respect to a desired behavior. The system is traditionally modeled as a labeled state-transition graph, and the behavior is specified by a temporal logic formula [1]. Early implementations, based on explicit-state model checking [2], suffered from the state explosion problem. The introduction of symbolic model checking with binary decision diagrams (BDDs) [3, 4] and other recently developed methods succeeded in partially overcoming this problem and enabled industrial applications of model checking for real-life systems, mostly in the hardware design industry. However, all these methods still suffer from the potential memory explosion problem on modern test cases. In this work we evaluate the application of Quantified Boolean Formulas (QBF) in bounded model checking (BMC) [14, 15] in an attempt to avoid the memory explosion problem.

In symbolic model checking the states of the state-graph are encoded by a vector of Boolean encoding variables; sets of states are represented by characteristic functions; transitions in the graph are represented with a transition relation: that is, a propositional formula referring to two sets of encoding variables. Properties are usually specified in a temporal logic. In this work we restrict our attention to safety properties – those that can be disproved by examining a finite computation path.

Symbolic model checking uses image computation to verify properties. Despite the increased capacity, compared to the explicit-state model checking, BDD-based techniques still suffer from the state explosion problem for models beyond a few hundred state variables, because BDDs do not always represent Boolean functions compactly.

In [5, 6], the authors evaluate SAT methods instead of BDDs for image computation. They use an explicit quantifier elimination to reduce the problem of reachability analysis to the problem of propositional satisfiability, and then apply SAT solvers to check the satisfiability. The explicit quantifier elimination, however, causes exponential blow-up in the size of the generated formulas in the worst case.

More recent papers [7-13] propose algorithms for SAT-based reachability analysis, where the quantifier elimination is implicitly implemented in the SAT solvers. The SAT solvers are modified such as to find all possible solutions rather than just one, by adding a blocking clause for each new solution found. Storing all the solutions in a compact data structure is a challenge, however. There have been attempts to use BDDs, zero-suppressed BDDs, or disjunctive normal form, but all of these are still of exponential size in the worst case.

Bounded Model Checking (BMC), introduced in [14, 15], is based on the representation of computation paths of a bounded length that falsify the property being checked. BMC with a specific bound k represents the paths of length k in the system by "unrolling" the transition relation k times, and examines whether the set of states falsifying the property is reached by these paths. The composed formula is sent to a SAT solver, and any satisfying assignment represents a counter-example of length k for the property. The use of BMC increased the capacity of model checkers to thousands of state variables, however, at a price: one no longer gets a fully certified answer to the verification problem, but rather an assurance that there are no counterexamples of a given length. To implement a complete model checking procedure the bound should be increased iteratively up to the length of the longest simple path in the system. Determining the sufficient bound for BMC is generally intractable, but it is exponential in the number of encoding variables in the worst case. Hence, the number of copies of the transition relation within the formulas being checked for validity increases from iteration to iteration up to an exponential number of times, leading, again, to a memory explosion for large systems and large bounds.

Induction based methods [16] provide another technique for estimating whether a bound is sufficient to ensure a full proof. Induction is particularly successful for local properties, but there are still many cases where the induction depth is exponential in the size of the model.

Lastly, in [17] the author uses Craig interpolation as an over-approximation technique for image computation aimed at reducing the number of iterations for a complete model checking procedure. The interpolants are obtained as a by-product of the SAT solver used to check BMC problems. This technique, like other techniques based on image computation, also suffers from a potential memory blow-up.

We present a QBF-based BMC method that does not perform unrolling of the transition relation and thus avoids the memory explosion problem. We also present a special-purpose QBF decision procedure for formulas used in this method.

The rest of the paper is organized as follows. Section 2 presents formulations of the bounded reachability analysis problem with propositional and quantified formulas. Section 3 describes our new algorithm for such an analysis; and Section 4 compares the performance of our algorithm to the usage of the existing SAT and QBF solvers in BMC. In section 5 we draw conclusions and suggest future directions.

2 Formulations of Bounded Reachability Checking Problem

Given a system $M=(S, I, TR)$, where S is the set of states, I is the characteristic function of the set of the initial states, and TR is the transition relation, the problem of reachability of the final states given by a characteristic function F in exactly k steps can be expressed in a number of ways.

As in classical BMC [14], the fact that the state Z_k is reachable from the state Z_0 in exactly k steps may be formulated by "unrolling" the transition relation k times:

$$R_k(Z_0, Z_k) = \exists Z_1, ..., Z_{k-1} : I(Z_0) \wedge F(Z_k) \wedge \bigwedge_{i=0}^{k-1} TR(Z_i, Z_{i+1}). \quad (1)$$

The validity of this formula may be proved or disproved by applying a SAT decision procedure on its propositional part.

Noticeably, the number of copies of the transition relation TR in this formula is the same as the number of steps being checked. When iteratively increasing the bound k, each successive iteration checks reachability of the final states in one more step than the previous iteration. Thus, for a complete check, the SAT procedure must be invoked on formulas containing an exponential number of copies of the transition relation.

To partially overcome the potential memory explosion, the following QBF formulation of the bounded reachability problem can be used:

$$R_k(Z_0, Z_k) = \exists Z_1, ..., Z_{k-1} : I(Z_0) \wedge F(Z_k) \wedge$$
$$\forall U, V : \left(\bigvee_{i=0}^{k-1} (U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1}) \right) \rightarrow TR(U, V). \quad (2)$$

Note that (2) contains only one copy of the transition relation. Increasing the bound, thus, would mean an addition of a new intermediate state and a term of the form $(U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1})$. Hence, the formula increase from iteration to iteration does not depend on the size of the transition relation, which is usually the biggest formula in the specification of the model.

The solution of (2) with a QBF solver usually requires a transformation of the quantifier-free part of the formula into a conjunctive normal form (CNF). Linear-time translation of a propositional formula to an equisatisfiable CNF formula [18] introduces artificial variables, resulting in a QBF with a $\exists\forall\exists$ quantifier prefix. The number of the universally quantified variables does not change in the QBF from iteration to iteration.

This approach to reachability checking partially solves the issue of formula growth, reducing the growth of the formula from iteration to iteration, but still requires an exponential number of iterations to fully verify the reachability.

To reduce the number of iterations, it is possible to apply "iterative squaring", similar to that used in BDD-based model checking [1]. This way each successive iteration checks the reachability of a final state in twice as many steps as the previous

iteration. Given a formula $R_{k/2}(X, Y)$ for checking reachability in $k/2$ steps, the following formula checks for reachability in k steps:

$$R_k(Z_0,Z_k)=\exists Z: I(Z_0)\wedge F(Z_k)\wedge\forall U,V: \\ \left[(U\leftrightarrow Z_0)\wedge(V\leftrightarrow Z)\vee(U\leftrightarrow Z)\wedge(V\leftrightarrow Z_k)\right]\rightarrow R_{k/2}(U,V). \quad (3)$$

The transition relation *TR* appears in (3) only once, as in the previously described technique. However, the number of universally quantified variables and the number of quantifier alternations grows from iteration to iteration.

This technique enables reducing the number of iterations to the number of the state encoding variables in the model, since it will then cover the worst-case diameter of the model. Note that not all bounds are checked by this technique, but the powers of two. It is possible, however, to overcome this problem by adding a self-loop to each state of the model, which would not change the reachability between states, but rather make (3) check reachability in *k or fewer* steps, instead of *exactly k* steps.

Using a bounded model checker to generate the three kinds of formulas mentioned above, we evaluated a few publicly available state-of-the-art DPLL [19] based SAT and QBF solvers, to check the feasibility of the QBF formulations of the reachability checking problem on a number of real-life industrial examples. Section 4 presents the details of the evaluation. Unfortunately, the general-purpose QBF solvers were unable to solve practically any of the formulas, while many of the corresponding propositional formulas were solved by the SAT solvers, often in a matter of seconds.

Noticeably, all the three kinds of formulas contain exactly the same information, but in different form. In (1) the formula explicitly contains the relation between any two successive states in the path from an initial state to a final one. Such an explicit representation often allows a solver to immediately restrict the choice of the next state in the path when the previous one has been chosen. For example, when in (1) the state Z_0 has been chosen by the algorithm, the Boolean constraint propagation (BCP) process [20] might possibly deduce the values of some variables in Z_1. Also, the choice of an impossible value for one of Z_1's variables could immediately cause a conflict. We may say that, in a sense, the SAT-based approach tries to examine for being final *only* the states within the set of states reachable from the initial ones.

In the QBF-based approaches this is not the case. The information about the relation between any two successive states is not found explicitly in the formula. Therefore, in formula (2) the DPLL-based solvers are unable to deduce anything about Z_1, when Z_0's value is set. This is because the relation between Z_0 and Z_1 is dependent on all possible choices of U and V. Additionally, a general-purpose DPLL-based QBF solver is restricted in the decision process to first set the values for the variables quantified in the outer level (Z_0, Z_1, etc.), before proceeding to the inner ones (U and V). Essentially, this means that the solver first chooses values for Z_0, Z_1, …, Z_k and only then checks whether such a choice constitutes a path in the model. In the solution of (3), not all states in the path have variables representing them in the formula, which further complicates the solution process.

3 jSAT Decision Procedure

Our motivation in this work was the inefficiency of the general-purpose QBF solvers demonstrated on formulas of form (2). To improve the efficiency of their solution, it is necessary to achieve a similar way of exploring the state space as a SAT solver on an equivalent propositional formula of form (1).

As in problems of form (2), jSAT holds in memory the encoding variables representing the states Z_0, Z_1, …, Z_k, U and V, but only holds the following propositional formula:

$$I(Z_0) \wedge TR(U,V) \wedge F(Z_k). \quad (4)$$

The states Z_i ($0 \le i \le k$) represent a path; the states U and V represent two neighboring states in that path. Instead of explicitly storing the fact that U and V represent a pair of neighboring states, as done in (2) with assistance of the terms of the form $(U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1})$, our algorithm implicitly assumes this information. The idea of the algorithm is to iteratively associate U and V with a pair of successive states, called the *current state* and the *next state*, until all states are decided. We call U and V *aliases*, since at each point of time during the algorithm execution they act as the states they are associated with.

jSAT is based on the classic DPLL algorithm [19, 20, 21] widely used in the current state-of-the-art SAT and QBF solvers. Given a CNF representation, DPLL algorithm iteratively chooses variables to assign a value to, as long as the partial assignment to the variables does not falsify the formula. When a partial assignment is found to falsify the formula, that is, when a conflict is discovered, the algorithm backtracks by unassigning some variables, and assigning an opposite value to one of them. The process by which the algorithm decides which variables to unassign and which variable to assign with an opposite value is called conflict analysis. The algorithm also incorporates an optimization called Boolean constraint propagation (BCP) or unit propagation [20], which aims at speeding up the search by making obvious decisions as soon as they can be made, based on the unit literal rule: if the current partial assignment causes all but one literal of a clause to have the value false, then the remaining literal must be assigned true in order not to falsify the clause and, consequently, the whole formula.

Most of the current state-of-the-art SAT and QBF solvers use additional optimization techniques, such as conflict-driven learning and non-chronological backtracking [20]. Our approach does not require any of above optimizations to be present, but we have implemented them, because they proved to provide a significant advantage. Numerous other optimization techniques exist that may also prove helpful in jSAT.

Intuitively, jSAT algorithm can be seen as a depth-first search in the state graph of the system from the initial states to the final ones. The algorithm starts by associating U with Z_0 and V with Z_1; thus the formula (4) becomes semantically equivalent to:

$$I(Z_0) \wedge TR(Z_0,Z_1) \wedge F(Z_k). \quad (5)$$

The states Z_0 and Z_1 are then chosen by finding an assignment to their encoding variables, if possible, so that Z_0 is an initial state and Z_1 is its successor. As soon as they are chosen, the algorithm makes Z_1 to be the current state and Z_2 to be the next

```
jSAT() {
    InitializeCurrentAndNextStates();
    while (true) {
        if (! SelectDecisionVariable()) {
            if (AllStatesDecided()) return true;
            if (! AdvanceCurrentState()) return false;
        }
        while (! BCP()) {
            if (! ResolveConflict()) return false;
        }
    }
}
```

Fig. 1. Pseudo-code of jSAT decision procedure

one: U becomes an alias to Z_1, and V becomes an alias to Z_2. The algorithm proceeds in this fashion until all states are successfully chosen, or until it discovers that such a choice is impossible. Whenever a conflict occurs, and the conflict analysis algorithm decides to unassign an encoding variable of the current state U, the search backtracks by setting U to be the previous chosen state and V to be the unsuccessfully chosen current state; it then tries to find another suitable such state.

The pseudo-code of the algorithm is shown on Fig. 1. The algorithm first initializes the states U and V to be associated with Z_0 and Z_1, respectively. The procedure SelectDecisionVariable() selects a still unassigned variable out of the encoding variables of the current state or, if all the encoding variables of the current state are assigned, from those of the next state. We restrict the decision strategy to selecting decision variables in the order of the states in the path: encoding variables of the state Z_0 are selected first, then the variables of Z_1, then the variables of Z_2, and so on. Such a restriction causes the algorithm to implement a depth-first search of the state graph and to "visit" only the states actually reachable from the initial states. The order of the selection of the encoding variables within one state is not important, and heuristics similar to the ones used in SAT/QBF solvers can be used.

SelectDecisionVariable() returns true if the decision is made successfully. Boolean Constraint Propagation is then performed by the procedure BCP(), which returns false in case of a conflict. If a conflict is produced, ResolveConflict() attempts to analyze it and backtrack to a previous decision level. In case the conflict cannot be resolved the algorithm terminates and the given formula is reported invalid.

SelectDecisionVariable() returns false whenever all the encoding variables of the current and the next states have been decided. If at this point all the states have been decided, as determined by the call to AllStatesDecided(), then a path has been found

```
ResolveConflict() {
    nBacktrackingLevel = AnalyzeConflict();
    if (nBacktrackingLevel < 0) return false;
    nFirstUndecidedPathState =
        Backtrack(nBacktrackingLevel);
    if (! RetractCurrentAndNextStates(
        nFirstUndecidedState)) return false;
    return true;
}
```

Fig. 2. Pseudo-code of jSAT decision procedure

from an initial state to a final one, and the algorithm terminates, reporting the given formula is valid. Otherwise, if undecided states remain, AdvanceCurrentState() advances U and V to the next pair of states by associating U with whatever was previously associated with V, and associating V with the next state in the path. During this operation new relations between the encoding variables become apparent. Thus, for example, when U and V are moved from the pair of states (Z_0, Z_1) to the next pair (Z_1, Z_2), the relations between the encoding variables of Z_1 and Z_2 become explicit in $TR(U, V)$. Since the newly discovered information may contradict some of the already made decisions, conflicts may arise during the adjustment operations. The procedure AdvanceCurrentState() returns false in case a conflict occurred that could not be resolved; in this case the algorithm terminates and the given formula is invalid.

Fig. 2 shows the pseudo-code of ResolveConflict() procedure. The call to AnalyzeConflict() checks whether the conflict is resolvable, and if yes, produces a conflict clause and returns the decision level to which to backtrack. Then, by the call to Backtrack(), the algorithm undoes the assignments made on the decision levels higher than the level to which the algorithm should backtrack. Backtrack() returns the earliest state among all the states, which does not have all its encoding variables assigned after the backtracking. If this earliest state is the one currently associated with U (i.e. is the current state) or an earlier one, U and V are retracted by RetractCurrentAndNextStates(), so that V is associated with the earliest undecided state in the path. This retraction implements the retreating step of the depth-first search in the state graph, so that the search is directed into another part of the graph. Noticeably, as with the operation of advancement of the current and the next states, the retraction may also produce conflicts, because the relations that were not explicit in the formula become explicit. RetractCurrentAndNextStates() returns false in case an irresolvable conflict occurred during the operation.

An important aspect of our algorithm follows from the fact that U and V represent different states at different points of time. It is therefore generally incorrect to produce learned conflict clauses that involve the encoding variables of U or V, or any artificial variable resulting from the translation of $TR(U, V)$ to CNF, as they will become use-

less as soon as U and V are adjusted to represent another pair of states. Therefore, the learned clauses must be formulated in terms of the encoding variables of Z_i. Our conflict analysis technique achieves this by using only decision variables in the learned clauses, somewhat similar to Last UIP learning scheme described in [27].

4 Experimental Results

We have implemented jSAT algorithm to measure its applicability to the problem of BMC. We used a bounded model checker to generate formulas of the forms (1), (2), (3) and (4). The formulas of form (1) were generated in DIMACS format and could be fed into many available SAT solvers. The formulas of forms (2) and (3) were generated in QDIMACS format and could be fed into the available QBF solvers. The formulas of form (4) were generated in a slightly customized DIMACS format, which adds the specification of the encoding variables to the formula description; our implementation of jSAT reads this modified DIMACS format.

We used a set of thirteen proprietary Intel® model checking test cases of different sizes to compare the run-time and memory consumption of the different BMC methods. For each test case we generated formulas of all kinds for the bounds in range from 3 to 20, resulting in the total amount of 234 formulas of each kind. Twenty of the formulas of forms (3) and (4) were publicly disclosed and participated in the QBF solver evaluation during SAT2004 conference. We used a dual Intel® Xeon™ 2.8 GHz Linux RH7.1 workstation with 4GB of memory for the experiments, and set a 600 second time out and 1 GB memory limits on all solvers.

We first used QuBE [24] and Semprop [25] QBF solvers to solve formulas of form (2) and (3) for all the test cases and to compare their run-time to the run-time of SAT solvers ([22], [23]) on the corresponding instances of form (1). As mentioned above, we discovered that both these solvers were able to solve only a few of the 234 formulas within the set time limits. This fact served as our motivation for the development of jSAT. We expected that jSAT, as a special-purpose decision procedure, would demonstrate memory consumption as low as the general-purpose QBF solvers, but a better run-time. We did not expect that jSAT run-time would be as good as that of the SAT solvers on the corresponding instances.

Our implementation is based on the single-threaded version of the SAT solver described in [22], which is reported to have slightly slower performance than zChaff [23]. To perform a fair analysis we chose to compare our algorithm to the solver [22], but also provide a comparison to zChaff II, as one of the best-known state-of-the-art solvers.

Table 1 shows the sizes of the test cases in terms of the state variables in the model, and the number of formulas each of the solvers successfully coped with (QuBE and Semprop have been omitted in this table for brevity). The numbers of solved SAT and UNSAT instances are shown separately. (We use the terms “SAT” and “UNSAT” in case of jSAT for consistency, even though jSAT solves a QBF. SAT result in this case means that the instance was proved valid; UNSAT means it was proved invalid.) Interestingly, jSAT’s results are especially close to those of [22] on SAT instances, where jSAT managed to solve 104 versus 106 instances solved by

Table 1. Number of instances solved by each solver per test case. There is a total of 18 instances in each test case, corresponding to BMC problems with bounds 3 to 20. SAT and UNSAT instances are shown separately. '-' sign specifies that there are no instances with the specific result for the corresponding test case

	# vars	jSat		[22]		zChaff	
		SAT	*UNSAT*	*SAT*	*UNSAT*	*SAT*	*UNSAT*
test08	10	-	16	-	18	-	18
test12	11	18	-	18	-	18	-
test10	12	-	18	-	18	-	18
test03	39	18	-	18	-	18	-
test06	160	-	1	-	12	-	18
test09	160	18	-	18	-	18	-
test05	199	-	0	-	18	-	18
test11	220	14	4	14	4	14	4
test04	626	0	1	4	2	13	2
test13	662	18	-	18	-	18	-
test02	914	-	0	-	13	-	18
test07	1055	0	-	11	-	17	-
test01	2013	18	-	5	-	11	-
Total (out of 234)		104	40	106	85	127	96
		144		191		223	

[22]. On UNSAT instances, the distance between jSAT and [22] is much more significant. Fig. 3 graphically shows the run-time performance of the solvers. The x-axis shows the number of instances solved, and y-axis shows the time taken to solve a particular instance; the curve is obtained by sorting the run-times in an ascending order. It is evident that jSAT significantly outperformed the general-purpose QBF solver QuBE. It still did not achieve the same run-times as the SAT solvers, though in the biggest test case test01 (see Table 1) it managed to solve in seconds all the instances, which required a much longer time for the other solvers. Also it is noticeable that on most of the instances that jSAT succeeded to solve the run-time achieved by jSAT is similar to that of the SAT solvers. However, jSAT performance degrades much faster than that of the SAT solvers when coming to the more complex instances: the slope of the performance curve of jSAT is much higher than that of the other solvers.

Fig. 4 graphically shows the memory consumed by jSAT, [22] and zChaff solvers when solving instances generated from the test case test13, which is the largest test case fully solved by all the three tools. The x-axis shows the BMC bound, and the y-axis shows the memory consumed when solving the corresponding instance. The run-time of jSAT on these instances varied from 1 to 3 seconds; the run-time of zChaff 1 to 6 seconds; and the run-time of [22] from 3 to 146 seconds. As expected, the graph indicates that jSAT memory consumption practically does not depend on the BMC bound being solved, while for SAT-based BMC approaches the memory consumption

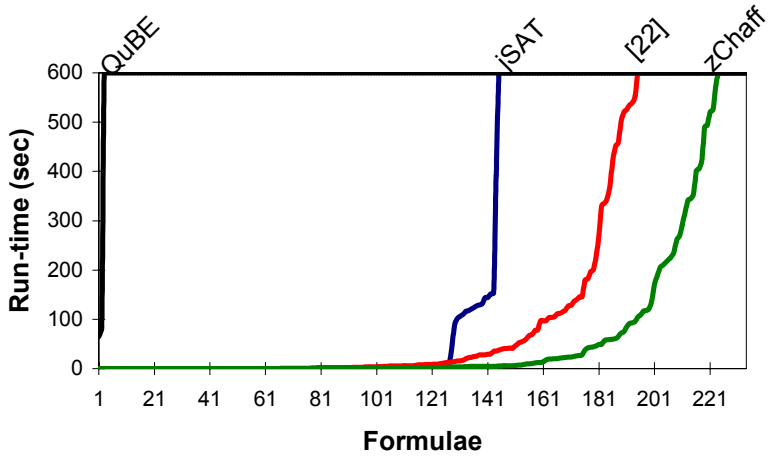

Fig. 3. Number of instances solved by each solver vs. the CPU time consumed

is proportional to the bound. The same behavior has been observed on the other test cases, including those that jSAT fails to complete.

The slower run-time of jSAT may be attributed to several factors. The decision heuristics of jSAT are different from those of [22] and zChaff, in that jSAT is restricted to assigning the state variables in order to achieve a depth-first search of the system state graph: the encoding variables of an earlier state on the path are assigned before the encoding variables of a following state. The order of selection among the encoding variables of the same state is not important, and in our implementation we used a variation of VSIDS decision heuristic [23] in which variable weights are not updated dynamically. We did not evaluate the performance of the SAT solvers with decision heuristics similar to the one of our jSAT implementation.

Another reason for the slower run-time of jSAT may be that our implementation did not use many of the advanced optimizations that are implemented in the other solvers. Compared to [22], our implementation of jSAT did not use restarting, did not remove conflict clauses and used a static decision heuristic.

The performance analysis of jSAT runs on the non-trivial instances showed that time is spent mostly on the current and the next state adjustment operations – those implemented by the procedures AdvanceCurrentState() and RetractCurrentAndNextStates() in Figs. 1 and 2. In our implementation we used a very simple method of adjusting the current and the next states:

- Unassign all variables, bringing the algorithm to its just-initialized state;
- Associate the current and the next state variables U and V with another pair of states;
- Re-assign all the state variables on the same decision levels as they were prior to the unassignment;
- Perform BCP.

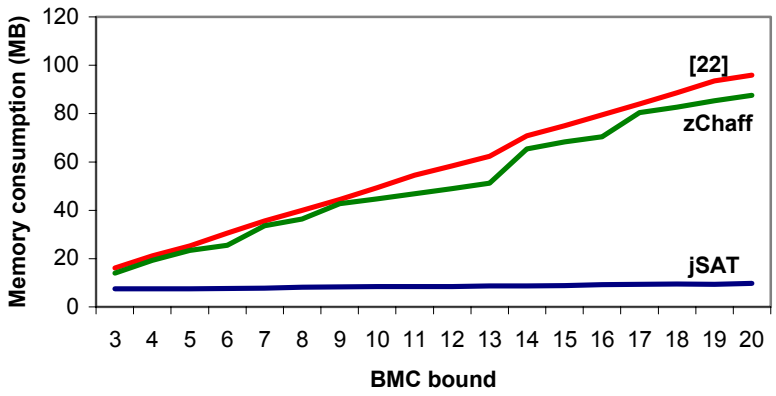

Fig. 4. Memory consumption of each solver on the instances generated for the test case test13

Our implementation used Watched Literals data-structure to represent the clause set of the instance being solved, and with this data-structure the state adjustment algorithm described above is quite simple but not very efficient. Indeed, in a highly connected system state graph the number of retreating steps performed during the depth-first search is very large; hence, the state adjustment operation is performed very frequently. Therefore, a more efficient approach to the state adjustment step is required, which is a subject for future research.

The performance analysis also showed that the conflict clauses built by our implementation systematically increase in length during the solution process. This follows from the way our implementation holds the implication graph [27] in memory (we used the same way as the implementation [22] on which we based ours): when the BCP process assigns a variable because of a clause becoming a unit, that variable is associated with the clause that implied it (the antecedent clause). The implication graph can be reconstructed by walking from a variable to the variables participating in its antecedent clause. If the antecedent clause belongs to the $TR(U, V)$ part of the formula, then the state adjustment operation may disassociate the variable and the clause, because after the adjustment of the current and the next states the encoding variables of U and V represent other variables. Thus, there is an information loss incurred by the state adjustment operation. This in turn causes more variables to be included in the learned conflict clauses, because the knowledge that some of the variables were implied by others is lost.

5 Conclusions

We have presented an evaluation of the usage of QBF in BMC, comparing the standard SAT-based BMC method to one using a QBF encoding of the problem. The usage of QBF in BMC avoids the memory explosion problem occurring with other model checking methods, because it does not require the "unrolling" of the transition relation. However, modern state-of-the-art general-purpose QBF solvers are still unable to handle the real-life instances of BMC problems in an efficient manner.

As the main contribution of our work, we presented a special-purpose QBF decision procedure, called jSAT, for the solution of QBF instances encoding BMC problems in form (2). A performance evaluation of our algorithm shows that it achieves the expected memory savings, and succeeds to solve significantly more instances than the general-purpose QBF decision procedures. Still, jSAT does not achieve run-times as short as the state-of-the-art SAT solvers on the corresponding SAT instances of the same problems, even though on some benchmarks it shows similar, and sometimes even much better, run-times.

We have identified a number of performance bottlenecks in our implementation of jSAT, so that a number of improvements can be made, and are subjects for the future research. These include:

- Data structures for efficient state adjustment operations;
- An alternative representation of the implication graph to avoid information loss incurred by the state adjustment operations; and
- Incorporation of additional optimization techniques used in the current state-of-the-art solvers.

References

[1] E. Clarke, O. Grumberg, D. Peled. "Model Checking". MIT Press, 2000.

[2] E.M. Clarke, E.A. Emerson, A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". ACM Transactions on Programming Languages and Systems, 8(2):244-263, 1986.

[3] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". IEEE Trans. Computers 35(8), 1986.

[4] K. L. McMillan. "Symbolic Model Checking". Kluwer Academic Publishers, 1993.

[5] P.A. Abdulla, P. Bjesse, N. Eén. "Symbolic Reachability Analysis Based on SAT Solvers". Tools and Algorithms for the Analysis and Construction of Systems (TACAS), 2000.

[6] P.F. Williams, A. Biere, E.M. Clarke, A. Gupta. "Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking". Computer-Aided Verification (CAV), 2000.

[7] O. Grumberg, A. Schuster, A. Yagdar. "Reachability Using Memory Efficient All-Solutions SAT Solver". Formal Methods in Computer-Aided Design (FMCAD), 2004.

[8] K.L. McMillan. "Applying SAT Methods in Unbounded Symbolic Model Checking". Computer-Aided Verification (CAV), 2002.

[9] H. J. Kang, I.-C. Park. "SAT-Based Unbounded Symbolic Model Checking". Proceedings of the 39th Conference on Design Automation (DAC), 2002.

[10] A. Gupta, Z. Yang, P. Ashar, A. Gupta. "SAT-Based Image Computation with Applica-

tion in Reachability Analysis". Formal Methods in Computer-Aided Design (FMCAD) 2000.
[11] P. Chauhan, E. M. Clarke, D. Kroening. "Using SAT Based Image Computation for Reachability Analysis". Technical Report CMU-CS-03-151, CMU, School of Computer Science, 2003.
[12] B. Li, M. S. Hsiao, S. Sheng. "A Novel SAT All-Solutions Solver for Efficient Preimage Computation". Design Automation and Test in Europe (DATE), 2004.
[13] M. Iyer, G. Parthasarathy, K.-T. Cheng "SATORI -- A Fast Sequential SAT Engine for Circuits". International Conference on Computer Aided Design (ICCAD), 2003.
[14] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. "Symbolic Model Checking Without BDDs". Tools and Algorithms for the Analysis and Construction of Systems (TACAS), 1999.
[15] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu. "Symbolic Model Checking Using SAT Procedures instead of BDDs". Proceedings of the 36th Conference on Design Automation (DAC), 1999.
[16] M. Sheeran, S. Singh, G. Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". Formal Methods in Computer-Aided Design (FMCAD) 2000.
[17] K.L. McMillan. "Interpolation and SAT-based Model Checking". Computer-Aided Verification (CAV), 2003.
[18] D. A. Plaisted, S. Greenbaum. "A structure-preserving clause form translation". Journal of Symbolic Computation 2 (1986), 293-304.
[19] M. Davis, G. Logemann, D. W. Loveland. "A machine program for theorem proving". Journal of the ACM, 394-397, 1962.
[20] I. Lynce, J. P. Marques-Silva. "An Overview of Backtrack Search Satisfiability Algorithms". 5th International Symposium on Artificial Intelligence and Mathematics, 1998.
[21] J. Gu, P. W. Purdom, J. Franco, B. W. Wah, "Algorithms for the Satisfiability (SAT) Problem: A Survey", URL: http://citeseer.ist.psu.edu/56722.html, 1996.
[22] Y. Feldman, N. Dershowitz, Z. Hanna. "Parallel Multithreaded Satisfiability Solver: Design and Implementation". Workshop on Parallel and Distributed Model Checking (PDMC), 2004.
[23] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik. "Chaff: Engineering an Efficient SAT Solver". Proceedings of the 38th Conference on Design Automation (DAC), 2001.
[24] QuBE QBF solver. URL: http://www.qbflib.org/~qube/.
[25] R. Letz. "Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas". TABLAUX, 2002. URL: http://www4.informatik.tu-muenchen.de/~letz/semprop/.
[26] D. Le Berre, l. Simon, A. Tacchella. "Challenges in the QBF Arena: the SAT'03 Evaluation of QBF Solvers". International Conference on Theory and Applications of Satisfiability Testing (SAT), 2003.
[27] L. Zhang, C. F. Madigan, M. H. Moskewicz, S. Malik. "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver". International Conference on Computer Aided Design (ICCAD), 2001.