Bounded Model Checking with QBF

Nachum Dershowitz[1], Ziyad Hanna[2], and Jacob Katz[2]

[1] School of Computer Science., Tel-Aviv University, Israel
nachumd@cs.tau.ac.il
[2] Intel Corporation, Haifa, Israel
{ziyad.hanna,jacob.katz}@intel.com

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[1]

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. Early implementations, based on explicit-state model checking, suffered from the state explosion problem. The introduction of symbolic model checking with binary decision diagrams (BDDs) and other recently developed methods, such as Bounded Model Checking (BMC), 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 BMC of safety properties in attempt to avoid the memory explosion problem. We also present a special-purpose purpose QBF decision procedure for a QBF encoding of BMC problems.

Assume 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. Let F be a characteristic function of the "bad" states violating the property being checked. As in classical BMC, the fact that a "bad" state Z_k is reachable from an initial state Z_0 in exactly k steps may be formulated by "unrolling" the transition relation k times:

[1] Due to space constraints references have been omitted in this text.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 408 – 414, 2005.

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

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:

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

The formula (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.

2 jSAT Decision Procedure

An experimental evaluation of general-purpose QBF solvers on formulas of form (2), presented in section 3, found them very inefficient, as they failed to solve practically any of the formulas in our test bench. This fact motivated the development of a special-purpose decision procedure, called jSAT, for formulas of this form.

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:

$$I(Z_0) \wedge TR(U,V) \wedge F(Z_k). \tag{3}$$

The states Z_i ($0 \leq i \leq k$) 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.

jSAT is based on the classic DPLL algorithm widely used in the current state-of-the-art SAT and QBF solvers. 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 (3) becomes semantically equivalent to:

$$I(Z_0) \wedge TR(Z_0,Z_1) \wedge F(Z_k). \tag{4}$$

```
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

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 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.

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 from an initial state to a final one, and the algorithm terminates, reporting the given formula is valid. Otherwise, if undecided states remain, AdvanceCurrentState()

```
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

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 useless 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.

3 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) and (3). The formulas of form (1) were generated in DIMACS format and could be fed into many available SAT solvers. The formulas of forms (2) were generated in QDIMACS format and could be fed into the available QBF solvers. The formulas of form (3) 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.

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		Base		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	

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. Some of the formulas of form (3) 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 state-of-the-art QBF solver to solve formulas of form (2) for all the test cases and to compare its run-time to the run-time of SAT solvers on the corresponding instances of form (1). We discovered that QuBE was 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.

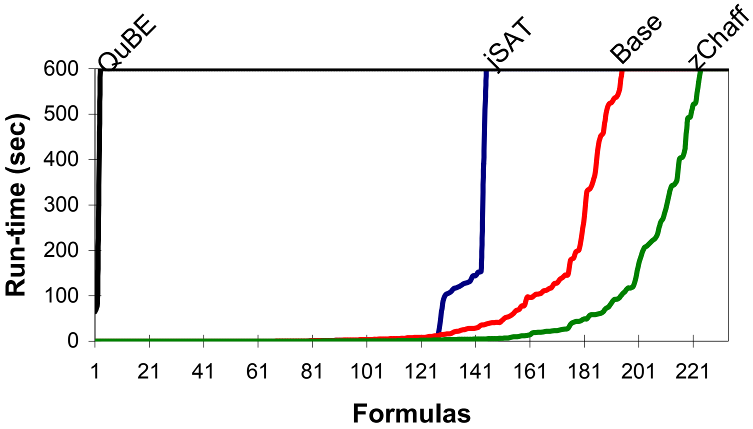

Fig. 3. Number of instances solved by each solver vs. the CPU time consumed

Our implementation is based on an existing solver[2] (base solver), which is reported to have slightly slower performance than zChaff. To perform a fair analysis we chose to compare our algorithm to that base solver, 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 has 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 the base solver on SAT instances, where jSAT managed to solve 104 versus 106 instances solved by the base solver.

On UNSAT instances, the distance between jSAT and the base solver 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 no-

[2] Y. Feldman, N. Dershowitz, Z. Hanna. "Parallel Multithreaded Satisfiability Solver: Design and Implementation". Workshop on Parallel and Distributed Model Checking (PDMC), 2004.

ticeable 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.

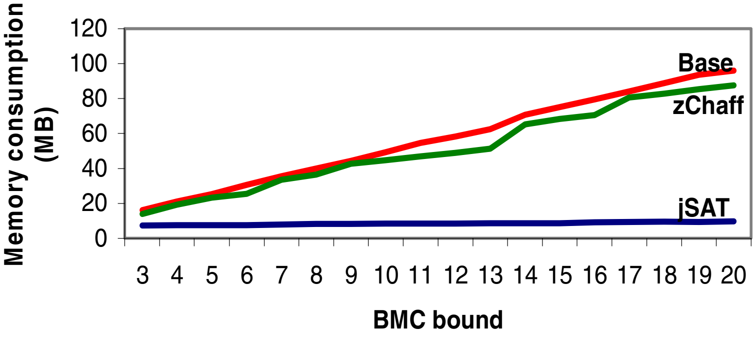

Fig. 4. Memory consumption of each solver on the instances generated for the test case test13

Fig. 4 graphically shows the memory consumed by jSAT, the base solver and zChaff 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 the base solver 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 is proportional to the bound. The same behavior has been observed on the other test cases, including those that jSAT fails to complete.

4 Conclusions

We have presented an evaluation of the usage of QBF in BMC, comparing a classical SAT-based BMC method to one using a QBF encoding of the problem, which avoids the memory explosion problem because it does not require the "unrolling" of the transition relation. We found that 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 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 a general-purpose QBF solver. 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 better, run-times.