Tel-Aviv University
Raymond and Beverly Sackler Faculty of Exact Sciences
The School of Computer Science

Model Checking with Quantified Boolean Formulas

A thesis submitted in partial fulfillment
of the requirements for the degree of Master of Science

by

Jacob Katz

The research work in this thesis has been carried out
under the supervision of Prof. Nachum Dershowitz

Novermber 2005

Abstract

Modern symbolic model checking techniques use Binary Decision Diagrams (BDD) and propositional satisfiability (SAT) decision procedures for checking validity and satisfiability of propositional Boolean formulas, which are used to encode sub-problems of symbolic model checking. Usage of propositional formulas imposes a potential exponential memory blow-up on the model checking algorithms due to the big formula sizes. Model checking 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. In this work, an evaluation of the usage of QBF in bounded model checking (BMC) is presented, using general-purpose SAT and QBF solvers. Additionally, a special-purpose decision procedure for QBF used in BMC is developed, and compared with the methods using general-purpose SAT and QBF solvers on real-life industrial benchmarks. The proposed decision procedure performs much better for BMC than general-purpose QBF solvers, without incurring the space overhead of propositional SAT.

Contents

Chapter 1

Introduction

During the last decades the field of automatic formal verification of concurrent finite-state systems has seen significant development. One of the primary domains for practical application of formal verification is in the digital hardware design industry. In the domain of digital hardware design, where the cost of correcting errors is extremely high, the task of validating design correctness prior to production is very important. One of the main means for design validation has traditionally been based on simulation. Simulation-based approaches are limited, however, due to the impossibility of exhaustive validation, since it is impractical to simulate all possible test vectors in modern designs with their ever-increasing size and complexity. Formal verification methods pose an alternative for simulation-based validation, but high-capacity automatic formal verification is still a challenge nowadays due to the inherent complexity of these methods.

Model checking [1], as one mean of formal verification, is a technique for the verification of the correctness of a finite-state system with respect to a desired behavior. Numerous works have been published on algorithms and methodologies for model checking, and it is also the focus of this work. The following sections briefly overview the background and the existing techniques for model checking, and then the scope of this work is described.

1.1. Background

Model checking is the process of determining whether a given formula is true in a given model. The model usually represents the program or the design being checked, and the formula is used to describe the desired program/circuit behavior, or specification, conformance to which it is necessary to prove.

A model is usually viewed as a finite Kripke structure, i.e. a labeled state transition system (a.k.a. state transition graph) where each state is labeled with a set of formulas holding in that state. The state of the system is defined as the set of values of the variables of the program represented by the model. A formula is usually specified in a temporal logic, such as LTL, CTL or CTL* [1]. The overall task of model checking is to compute the set of states in which the given specification holds. Normally some states of the system being checked are designated as initial states, and the system is said to satisfy the specification if all its initial states are a subset of the set of states in which the specification holds.

1.1.1. Explicit state model checking

Explicit state model checking algorithm [1][2] operates on the Kripke structure explicitly represented in memory as a collection of vertices and edges. The CTL model checking algorithm iteratively processes the given formula starting from the innermost sub-formulas, and proceeds by marking every state in the Kripke structure with the sub-formulas holding in those states. A linear algorithm exists for finding states in which a CTL formula holds, given that the states are already marked with where the sub-formulas hold. Thus, the overall algorithm has the complexity $O(|\varphi|\cdot|M|)$, where φ is the CTL formula being checked and M is the Kripke structure. The algorithm can also be extended to handle fairness constraints, as explained for example in [1][2], having an overall complexity $O(|\varphi|\cdot|M|\cdot|F|)$, where F is the set of fairness constraints.

An LTL model checking algorithm using a tableau is described in [1][44]. A tableau is a graph constructed from the given Kripke structure and the sub-formulas of the given formula, so that the given formula is true in the given model if and only if there is a computation in the structure that is also a path in the tableau. The complexity of the algorithm is $O(|M|\cdot 2^{|\varphi|})$, i.e. linear in the size of the model and exponential in the size of the formula.

CTL* model checking [1] has the same complexity as the one for LTL, and usually combines the ideas of CTL and LTL model checking. The tableau-based LTL model checking algorithm can be extended to the cases where the sub-formulas of an LTL formula are arbitrary state formulas (including CTL formulas, which are always state formulas), and not only propositions.

Because of the explicit representation of the state graph, explicit state model checking is unable in practice to handle state-graphs of real-life systems, which have at least thousands, and often millions, of states. For example, in digital hardware verification a state of the system usually corresponds to the values stored in the system's binary storage elements (latches and flip-flops) at specific moment in time; thus a system with N such elements is represented by a state-transition graph with 2^N vertices in the worst case, i.e. has an exponential memory complexity. This is a so-called "state explosion problem", which is inherent in the explicit state model checking. This state explosion problem is one of the main factors limiting the industrial application of model checking techniques, in general, and explicit state model checking, in particular. A number of techniques to work around this problem to some extent have been developed.

1.1.2. Symbolic model checking

Symbolic model checking [4][40] evolved as an approach to handle the state explosion problem faced in explicit state model checking. Since the model explicitly represented as a Kripke structure is inherently big, the main idea of symbolic model checking is to avoid explicit construction of the model; instead, the model is represented symbolically with Boolean functions, originally using Binary Decision Diagrams (BDD) [3] for that purpose.

In symbolic model checking states of a Kripke structure are encoded using a vector of Boolean variables, the so-called *state encoding variables*. N variables are sufficient to encode 2^N states. With such an encoding it is possible to construct a formula over those variables so that it is true in a single state – this formula characterizes the state, and is called the *characteristic function of the state*. Similarly, a set of states can be characterized by a formula which is true in all states of the set, and false in all other states – this is called the *characteristic function of the state set*. Transitions in the Kripke structure may be characterized by Boolean formula over two copies of state variables – current state variables and next state variables. It is possible to construct a formula over current (V) and next state (V') variables, called the *transition relation* of the system, which is true if and only if there is a transition in the Kripke structure from the state represented by V to the set represented by V'.

The forward image of a state set S is defined as the set of states reachable from any state in S along a single transition. Given a characteristic function for S and the transition relation TR of the system, the characteristic function of the image of S can be expressed by:

$$\text{Image}(V') = \exists V : S(V) \wedge TR(V,V') \tag{1}$$

Similarly, the inverse image of a state set S is a set of states from which any state in S can be reached along a single transition. The inverse image of S can be expressed by:

$$\text{Image}^{-1}(V) = \exists V' : S(V') \wedge TR(V,V') \tag{2}$$

An iterative computation of the forward image allows the computation of a set of states reachable from a state set S along any number of transitions in the following way:

$$\begin{aligned} S_0 &= S \\ S_{i+1} &= S_i \vee \text{Image}(S_i) \end{aligned} \tag{3}$$

This iterative computation may be stopped whenever $S_{i+1} \leftrightarrow S_i$, meaning that a fix-point is reached and the set S_{inf} of all reachable states has been computed.

The symbolic model checking procedure for CTL, described in [1][4][40], is based on the fix-point characterization of CTL operators, and is derived from the iterative fix-point algorithm for symbolic model checking for the μ-calculus using translation of CTL formulas to μ-calculus. Since every CTL formula is a state formula (is either true or false in every state), it can be thought of as characterizing a set of states. CTL operators applied on formulas P and Q can be syntactically translated directly into Boolean connectives on characteristic functions of P and Q, into image calculation, or into fix-point computations, as follows:

- $\neg P, P \vee Q, P \wedge Q$ are translated directly into Boolean connectives
- $\mathrm{EX}P$ is translated to the inverse image of P: $\mathrm{Image}^{-1}(P)$
- $\mathrm{E}(Q \mathrm{U} P)$ is translated to $\mu Y : \left(P \vee (Q \wedge \mathrm{EX}Y)\right)$ (4)
- $\mathrm{EG}P$ is translated to $\nu Y : \left(P \wedge \mathrm{EX}Y)\right)$

All the rest of the operations can be viewed as abbreviations of the above.

Having such a translation of CTL operators, the model checking goal is achieved by computing the set of states *R* in which the given formula holds and checking whether that set of states contains the set *I* of initial states, by checking validity of the formula

$$\forall V : I(V) \rightarrow R(V) \tag{5}$$

where *V* is the vector of state variables, *I*(*V*) is the characteristic function of the set of initial states, and *R*(*V*) is the characteristic function of the set of states where the specification holds.

Symbolic model checking for LTL is based on the one for CTL and the method of tableau construction, similary to the explicit state model checking algorithm.

With the introduction of symbolic model checking, the capacity of model checkers increased dramatically, and enabled industrial applications of model checking, mainly in hardware verification domain. Still, the memory blow-up problem has not been totally resolved, as will be described later in this text.

1.1.3. Boolean satisfiability problems in symbolic model checking

There are two places in symbolic model checking where the validity of Boolean formulas should be determined:

- in the test for the convergence of fix-point computations; and
- in the test whether the property holds, made in the end or in the process of the fix-point computations.

The termination condition in fix-point computations has the form $R_i(V) \leftrightarrow R_{i+1}(V)$, where R_i is the set of states in iteration *i*, and R_{i+1} is the set of states in iteration *i+1*. In fact, it is sufficient to check $R_i(V) \rightarrow R_{i+1}(V)$ or $R_{i+1}(V) \rightarrow R_i(V)$, depending on the kind of the fix-point being computed (greatest fix-point or least fix-point). Both these conditions have the form $R_i(V) \to R_j(V)$. In order to check this fact, there is a need to prove that $\forall V : R_i(V) \to R_j(V)$ is valid or, alternatively, that its negation $\exists V : \neg\left(R_i(V) \to R_j(V)\right)$ is unsatisfiable. Noticeably, checking this requires handling of a proper Quantified Boolean Formula (QBF), since there are existentially quantified variables under a negation resulting from the computation of R_i and R_j.

The test for the property holding in the model is performed by checking that the set of the initial states is contained within the set of states satisfying the specification. To accomplish that one needs to check the validity of $\forall V : I(V) \to R(V)$, where V is the vector of state variables, $I(V)$ is the characteristic function of the set of the initial states, and $R(V)$ is the characteristic function of the set of states where the specification holds; or unsatisfiability of its negation $\exists V : \neg\left(I(V) \to R(V)\right)$. This is also a pure QBF, since $R(V)$ contains existential quantifiers, although in some cases this latter check can be simplified with a number of simpler checks made during the fix-point iterations.

1.1.4. Coping with QBF

Despite the need to evaluate QBF, none of the currently existing model checking techniques operates directly on QBF with any sort of decision procedure, because no practically efficient decision procedure for QBF has been developed. Essentially, model checkers resort to using propositional formulas in one form or another instead of QBF. Though both quantified and propositional representations have the same expressiveness, QBF are usually much more succinct. Hence, the usage of propositional representation results in a significant memory overhead.

One of the simplest (and most widely described) methods for coping with the presence of QBF is quantifier elimination. Quantifier elimination can be seen as a process of syntactic replacement of quantified formulas with propositional ones, based on the following rules:

$$\begin{aligned} \exists x : f(x) &\equiv f(x \mapsto 0) \vee f(x \mapsto 1) \\ \forall x : f(x) &\equiv f(x \mapsto 0) \wedge f(x \mapsto 1) \end{aligned} \tag{6}$$

The main drawback of this approach is that it can result in an un-quantified formula that is exponentially larger than the original quantified representation. When formulas are represented with Ordered Binary Decision Diagrams (OBDD), quantifier elimination can be performed according to the rules in (6) as a combination of restriction and logical-or/and operations. In [1] the authors also describe a more efficient algorithm to perform quantifier elimination in some cases.

In [6] the authors use Binary Expression Diagrams (BED) for representation of Boolean functions, and provide a technique to eliminate quantifiers in this representation. BED is a DAG representation, which is not canonical, unlike OBDD. In order to check the satisfiability of a formula, they either translate a BED representation to OBDD, or to conjunctive normal form (CNF) in order to use a SAT solver.

In [5] the authors proposed another representation for Boolean formulas, namely Reduced Boolean Circuit (RBC), which is somewhat similar to BED, and a quantifier elimination technique similar to that of [6]. They then feed the produced formula to a SAT solver in order to check the satisfiability.

Both [5] and [6] implement optimizations for some simple and common cases of quantifier elimination, but still do not overcome the exponential space complexity in many cases.

More recent works [7][8][9][10][11][12][42] propose algorithms for SAT-based reachability analysis, where the quantifier elimination is implicitly implemented in the SAT solvers. The SAT solvers are modified so as to find all possible solutions rather than just one, e.g. 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 an exponential size in the worst case, compared to the quantified representation.

It should be noted at this point that the performance comparisons of SAT and BDD-based techniques in symbolic model checking showed that in many cases SAT-based techniques outperform BDD-based ones, and in many cases the opposite is true. Thus, it is commonly believed that SAT and BDD-based techniques complement each other.

Bounded model checking (BMC), introduced in [14][15], is a model checking technique not based on fix-point computations. The authors show that BMC can be used for symbolic model checking of LTL without a tableau construction. BMC places a bound on the length of the non-looping prefix of the infinite paths checked to satisfy the property and attempts to falsify the property within the bounded model. In a single run of the algorithm all infinite paths (starting in an initial state) with prefix of length *k* are checked. By indefinitely incrementing the bound *k* all possible paths can be checked, resulting with a complete model checking procedure.

The problem of checking all paths with a prefix of length *k* is reduced to checking satisfiability of a propositional formula built up from the transition relation “unrolled” *k* times and the formula specifying that there is a loop in the path from the last state to any other state in the prefix. The authors showed that there is a polynomial time algorithm for translation of an LTL formula to the propositional formula to be checked for satisfiability. A satisfying assignment to this formula constitutes a counterexample of length *k* for the property.

Bounded model checking has been shown very efficient for *bug hunting*, i.e. the verification process aiming at quick finding of counterexamples, as opposed to producing a statement of conformance of the model to the specification. It has been shown that bounded model checking significantly outperforms classical symbolic model checking in many cases. Specifically, it is able to find short counterexamples very efficiently, since it only requires a propositional satisfiability check for a small bound *k*, and not a full blown iterative computation of fix-points. Although bounded model checking may be performed with any method for propositional satisfiability checking, including BDD, in [41] the authors showed that SAT-based approach clearly outperforms a BDD-based one.

In [14] the authors showed that for a complete model checking procedure, the BMC bound *k* does not need to be increased indefinitely, but only up to the *recurrence diameter* of the system: the length of longest simple path between two states. It is known that in many cases even a lower upper bound exists, but determining an optimal bound is, however, generally intractable. In the worst-case the bound is exponential in the number of the encoding variables of the system. Hence, the number of copies of the transition relation within the formulas checked for satisfiability increases from iteration to iteration up to an exponential number of times, leading to a memory explosion for large systems and bounds.

In [17] the authors proposed a technique for reducing the number of iterations required for a complete model checking procedure based on BMC. This method uses Craig interpolation as an over-approximation technique for image computation. The interpolants are obtained as a by-product of the SAT solver used to check BMC problems by analyzing their proofs of unsatisfiability. The number of iterations required for this method to complete is exponentially smaller than for the classical BMC, and it has been shown to outperform all the other model checking methods in many cases. However, although based on the idea of bounded model checking, this method relies on image computation, and therefore also suffers from the potential memory explosion, since the computed interpolants can be of exponential size.

Lastly, there are model checking methods based on induction, e.g. [16]. In these methods the induction hypothesis contains an "unrolling" of length *k* of the transition relation (also called the induction depth), similarly to BMC. An inductive proof of a property may require the induction depth to be exponentially big, resulting in a memory explosion, just like in regular BMC.

1.2. Scope of this work

As evident from the previous section, practically all modern model checking techniques suffer from a potential memory explosion problem. In the explicit state model checking this is due to the enormous size of the system state graph, while in the symbolic model checking techniques this is due to the usage of propositional formulas, which are extremely non-succinct, in places where the validity of a QBF must be determined.

Determining validity of a QBF is possible without translation to a propositional formula. Recently a significant effort has been invested in attempt to produce an efficient decision procedure for generic Quantified Boolean Formulas, following the success of SAT solvers for propositional formulas. However, no publicly available solver currently shows a reasonable run-time performance on existing QBF benchmarks, which include random, hand-made and a few industrial examples. There is also no known application of QBF solvers to the problem of model checking.

In this work the attention is restricted to model checking of safety properties, i.e. properties that can be disproved by examining finite computation paths. Liveness properties can be reduced to safety, for example by the method of [43]. Moreover, the attention is restricted to bounded model checking.

The main contribution of this work includes:

- the evaluation of QBF encodings of BMC problems on real-life industrial test cases, and the performance analysis of the best-to-date publicly available QBF solvers on those encodings;
- a special-purpose QBF decision procedure, which is able to solve QBF encodings of BMC problems faster than the general-purpose QBF solvers.

The rest of this document is organized as follows: Chapter 2 presents the QBF-based approach to bounded model checking and its evaluation on a set of real-life test cases. Chapter 3 describes the principles of SAT decision procedures, then Chapter 4 overviews their extension for QBF. In Chapter 5 the special-purpose QBF decision procedure is presented together with experimental results. In Chapter 6 some conclusions are drawn and future directions are presented. Chapter 7 surveys related work.

Chapter 2

Bounded model checking with QBF

One of the primary goals of this work is to evaluate the applicability of QBF formulations of BMC problems and compare their usage to the SAT-based approach on the corresponding SAT formulations. The following sections present the QBF formulations that were evaluated, the method by which they were generated, and the comparison procedure to the classical SAT-based method.

In the following, let us 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. BMC of safety properties assumes a set of so-called “bad” or “final” states, i.e. the states that violate the property, given by a characteristic function *F*. The problem of BMC with a specific bound *k* is then to determine whether a state in *F* can be reached from a state in *I* in exactly *k* transitions.

2.1. Formulations of BMC problems

In classical SAT-based 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:

$$R_k(Z_0, Z_k) = \exists Z_1, \ldots, Z_{k-1} : I(Z_0) \wedge F(Z_k) \wedge \bigwedge_{i=0}^{k-1} TR(Z_i, Z_{i+1}) \tag{7}$$

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 (7) 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, and hence the memory explosion incurred by this BMC method.

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

$$\begin{aligned} R_k(Z_0,Z_k) = \exists Z_1,\dots,Z_{k-1} : I(Z_0) \wedge F(Z_k) \wedge \\ \forall U,V : \left(\bigvee_{i=0}^{k-1} (U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1}) \right) \rightarrow TR(U,V) \end{aligned} \tag{8}$$

Formula (8) contains only one copy of the transition relation. Increasing the bound in this case would mean the addition of a new intermediate state and a term of the form $(U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1})$. Thus, the formula size 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 (8) with a QBF solver usually requires 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, e.g. by the method of [18], introduces artificial variables, resulting in a QBF with an $\exists\forall\exists$ 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 [40]. 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:

$$\begin{aligned}&R_k(Z_0,Z_k) = \exists Z : I(Z_0) \wedge F(Z_k) \wedge \forall U,V :\\ &\left[(U \leftrightarrow Z_0) \wedge (V \leftrightarrow Z) \vee (U \leftrightarrow Z) \wedge (V \leftrightarrow Z_k)\right] \rightarrow R_{k/2}(U,V)\end{aligned} \tag{9}$$

The transition relation *TR* appears in (9) 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 the reduction of 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, however, that not all bounds are checked by this technique, but only powers of two. It is possible to overcome this problem by adding, for example, a self-loop to each state of the model, which would not change the reachability between states, but rather make (9) check reachability in *k or fewer* steps, instead of *exactly k* steps.

2.2. Benchmarking

To measure the feasibility of different formulations of BMC problems, a test suite of real-life industrial model checking examples was created. Formulas of the three forms described above were generated, representing BMC problems for different bounds. Run-time and memory consumption measurements were carried out on the generated formulas using publicly available SAT and QBF solvers.

2.2.1. Test generation

The tests in the test bench used in the benchmarking were created out of thirteen proprietary Intel® model checking examples. The process of test generation is depicted in Fig. 1. The models were extracted using a proprietary Intel® model checker from hardware design descriptions with embedded property specifications. The code of the Intel® model checker was augmented to dump out the following information after it built the model internally:

1. the state encoding variables;
2. the characteristic function of the initial states;

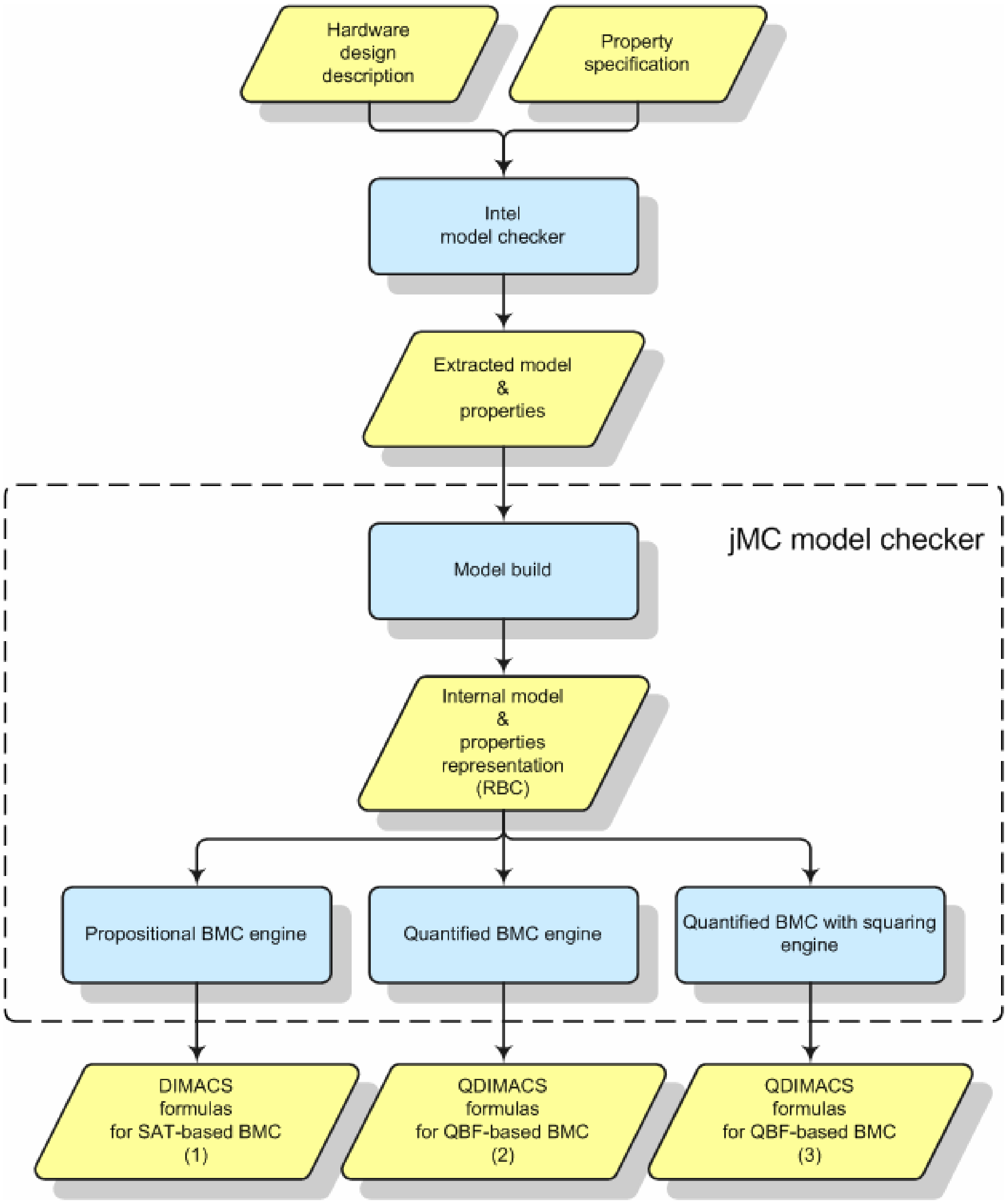

Hardware design description
Property specification
Intel model checker
Extracted model & properties
jMC model checker
Model build
Internal model & properties representation (RBC)
Propositional BMC engine
Quantified BMC engine
Quantified BMC with squaring engine
DIMACS formulas for SAT-based BMC (1)
QDIMACS formulas for QBF-based BMC (2)
QDIMACS formulas for QBF-based BMC (3)

Fig. 1. The process of test generation for benchmarking SAT and QBF solvers on BMC problems. Three different kinds of formulas are generated for every test case and bound.

3. the transition relation of the system;
4. the invariant characterizing the assumptions for the model checker; and
5. the characteristic function of the "bad" states violating the property.

The extracted models were saved in files in a simple format, described in Annex A. A simple model checker, called jMC, was developed, which was used to read the extracted models and generate formulas of forms (7), (8) and (9) in DIMACS (for SAT-based BMC) and QDIMACS (for QBF-based BMC) formats. In jMC the model and the properties were read from a file into an internal data-structure. The propositional formulas were stored in a Reduced Boolean Circuit (RBC) data structure [5], which allows a compact but non-canonical representation due to high sharing of common sub-formulas. Several engines were developed implementing different BMC approaches, each producing a set of formulas to be fed into either SAT or QBF solvers:

1. Propositional BMC engine: a classical SAT-based approach using "unrolling" of the transition relation, producing propositional formulas of form (7) in CNF in DIMACS format.
2. Quantified BMC engine: a QBF-based approach producing formulas of form (8) in CNF in QDIMACS format.
3. Quantified BMC with squaring engine: a QBF-based approach producing formulas of form (9) in CNF in QDIMACS format.

Later on, an additional engine was implemented to generate formulas for the special-purpose decision procedure developed as the main part of this work. This is described in Chapter 5.

For every one of the thirteen test cases, formulas of the three kinds were generated for a number of iterations required to perform BMC for bounds in the range 3 – 20. For formulas of forms (7) and (8) eighteen instances were generated for each test case, resulting in the total amount of 234 formulas of each kind; for formulas of form (9) four instances were generated for each test case, corresponding to bounds 4, 8, 16 and 32. Table 1 shows the sizes of the models in terms of the number of state variables and the sizes of the generated formulas for BMC with bounds 3 and 20 in terms of the number of variables and clauses in the CNF encoding. Note that the formulas for SAT-based BMC are extremely big and even for moderate examples have hundreds of thousands of

	# state	SAT-based BMC				QBF-based BMC				QBF-based BMC with squaring			
		Bound 3		Bound 20		Bound 3		Bound 20		Bound 3		Bound 32	
	vars	Vars	Clauses	Vars	Clauses	Vars	Clauses	Vars	Clauses	Vars	Clauses	Vars	Clauses
test08	10	298	811	1,896	5,265	294	783	1,144	3,163	354	923	684	1,763
test12	11	115	268	744	1,747	248	633	1,183	3,251	314	787	677	1,711
test10	12	469	1,303	3,036	8,545	387	1,039	1,407	3,895	459	1,207	855	2,215
test03	39	629	1,555	4,012	10,021	960	2,470	4,275	11,752	1,194	3,016	2,481	6,292
test06	160	7,683	21,988	49,452	143,878	5,656	15,488	19,256	53,568	6,616	17,728	11,896	31,168
test09	160	7,296	20,746	46,889	135,564	5,530	15,078	19,130	53,158	6,490	17,318	11,770	30,758
test05	199	9,945	28,378	65,535	187,957	7,123	19,355	24,038	66,717	8,317	22,141	14,884	38,857
test11	220	8,925	25,297	56,967	164,748	7,235	19,671	25,935	72,031	8,555	22,751	15,815	41,231
test04	626	109,379	322,513	725,629	2,144,624	48,465	138,821	101,675	287,809	52,221	147,585	72,879	200,169
test13	662	39,955	114,643	259,306	753,401	26,121	71,933	82,391	229,489	30,093	81,201	51,939	136,809
test02	914	33,548	96,463	223,081	641,942	28,582	77,075	106,272	294,607	34,066	89,871	64,228	166,647
test07	1055	52,805	152,260	340,819	997,959	37,993	104,461	127,668	355,551	44,323	119,231	79,138	207,851
test01	2013	270,051	792,118	1,780,280	5,250,606	128,859	366,227	299,964	845,321	140,937	394,409	207,366	563,501

Table 1. Quantitative description of test cases used in benchmarking. For every test the number of state variables is shown, as well as the sizes of formulas produced by jMC for bounds 3 and 20 for the three BMC methods.

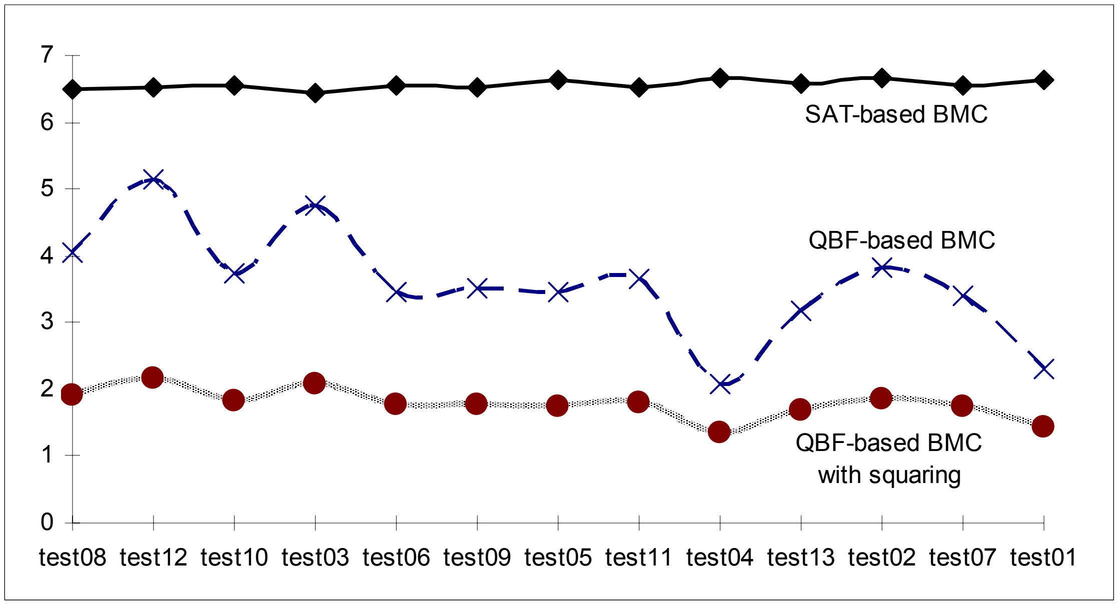


Fig. 2. Ratio of the number of clauses in a formula for BMC bound 20 relative to the formula of the same kind for BMC bound 3.

clauses. It is expected that the formula size in SAT-based BMC grows linearly with the bound, and indeed Fig. 2 shows that the formulas for bound 20 are 6-7X bigger than for bound 3 regardless of the size of the model. In the QBF-based approach, the ratio of the number of clauses in instances for bounds 20 and 3 is not constant, and in both approaches follows the same pattern, though with different amplitude. The peaks on the curves appear in test cases where the transition relation is relatively small, and thus the overhead of the other parts of the formula is significant. The falls, on the other hand, correspond to test cases with a big transition relation, where the additional terms added to the formula do not affect its size significantly.

Twenty of the formulas of forms (8) and (9) were publicly disclosed and participated in the QBF solver evaluation during the International Conference on Theory and Applications of Satisfiability Testing (SAT) in 2004.

2.2.2. Benchmarking environment

The formulas generated as described above were used to measure and compare the run-time and memory consumption of publicly available SAT and QBF solvers. The environment and software used to perform the measurement were reused with minor modifications from [45].

Two SAT solvers were used to solve formulas of form (7): the solver described in [22][45] and zChaff II [52], which is one of the best state-of-the-art SAT solvers available at the time of comparison. To solve QBF formulas of forms (8) and (9) QuBE [24] and Semprop [25] QBF solvers were used, which are both state-of-the-art solvers available at the time of comparison.

A dual Intel® Xeon™ 2.8 GHz Linux RedHat 7.1 workstation with 4GB of memory was used for the experiments. Each solver was limited to solve every single instance within 10 minutes run-time and 1GB memory envelope.

2.2.3. Benchmarking results and analysis

The summary of run-time evaluation results is presented in Table 2. Detailed results are provided in Annex C.

As can be seen, general-purpose QBF solvers were unable to solve practically any of the formulas of form (8), while many of the corresponding propositional formulas were solved by the SAT solvers, often in a matter of seconds. In fact, QuBE was able to solve a few instances, but its results were unstable, and highly sensitive to insignificant and seemingly irrelevant changes in the encoding of the instances during the experiments, such as changing the numbering of literals, for example. The quality of QuBE results is not clear, therefore. None of the instances of form (9) were solved by any of the QBF solvers.

Noticeably, all the three kinds of formulas contain exactly the same information, but in different form. In (7) 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 (7) 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 variables could immediately cause a conflict. It is possible

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

Table 2. Summary of the evaluation results of two SAT and two QBF solvers on BMC problems of forms (7) and (8), respectively. The numbers of solved SAT and UNSAT instances are shown separately. A '-' sign specifies that there are no instances with the specific result for the corresponding test case. (The terms "SAT" and "UNSAT" are used for QBF for consistency. A SAT result for a QBF means that the instance was proved valid; UNSAT means it was proved invalid.)

to 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 (8) the DPLL-based solvers are unable to deduce anything about Z_1, when Z_0 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 (9), not all states in the path have variables representing them in the formula, which further complicates the solution process.

The inefficiency of the general-purpose QBF solvers on the QBF formulations of BMC problems served as motivation for the development of a special-purpose QBF decision procedure for these specific kinds of QBF, which is described later in this dissertation. Formulas of forms (8) and (9) have very specific structure, which might possibly enable a more efficient special-purpose algorithm for their solution. The scope of the development was limited to formulas of form (8); and the goal for the developed special-purpose solver was to improve the run-time performance of the solution to come as close as possible to SAT-based approaches on the corresponding SAT formulations, and still preserve the memory efficiency enabled by QBF formulations.

Chapter 3

Propositional SAT

The problem of propositional satisfiability (SAT) is the problem of determining whether a given Boolean propositional formula contains a contradiction, and if it does not, finding a satisfying assignment for it. If the formula has a satisfying assignment, then it is said to be satisfiable, otherwise it is said to be unsatisfiable.

Algorithms for the solution of SAT problem (SAT solvers) have seen major advances in recent years. A SAT solver competition is now held annually within the framework of the International Conference on Theory and Applications of Satisfiability Testing (SAT).

Excellent surveys on the techniques used in modern SAT solvers can be found in [20][21][28]. Here only a brief overview is given.

SAT algorithms can be divided into two groups: complete and incomplete. Complete methods always produce an answer whether the given formula is satisfiable or not. Incomplete methods (e.g. local search methods), on the other hand, can only find a satisfying assignment, but are unable to produce an answer when the given formula is unsatisfiable.

Modern most successful SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [19] – a complete backtrack search method. In this work only DPLL-based complete backtrack search methods are covered.

3.1. Formula representation

DPLL-based SAT solvers operate on formulas in a conjunctive normal form (CNF). In this form the formula is represented as a conjunction of clauses. A clause is a disjunction of literals. A literal is an occurrence of a variable (positive literal) or the negation of a variable (negative literal). For example, the CNF formula $(x_1 \vee \neg x_2) \wedge (\neg x_3) \wedge (x_1 \vee x_3)$ contains three clauses $(x_1 \vee \neg x_2)$, $(\neg x_3)$, $(x_1 \vee x_3)$, two positive literals x_1, x_3 and two negative literals $\neg x_2$, $\neg x_3$.

For a variable assignment to satisfy a CNF formula, it must satisfy every one of the formula's clauses separately. If at least one of the clauses is unsatisfied by the variable assignment, the whole formula is unsatisfied.

There are polynomial time algorithms that convert an arbitrary propositional formula into a CNF formula having an equivalent satisfiability (equisatisfiable formula) [18]. This is done by the addition of extra variables, one for each non-atomic sub-formula, and expressing the equivalence of the extra variable with that sub-formula in CNF.

3.2. Basic DPLL algorithm

The DPLL algorithm is a complete backtrack search procedure for determining the satisfiability of propositional formulas in CNF representation. The algorithm performs a systematic search for a variable assignment satisfying the formula. It starts with an empty partial variable assignment and incrementally extends it by assigning variables one after another as long as the partial assignment does not falsify the formula. If the partial assignment has been successfully extended to a full one, the algorithm terminates and the formula is reported satisfiable with the found variable assignment. Whenever a partial assignment falsifies the formula, i.e. a conflict is discovered, the algorithm backtracks by flipping the value of the most recently assigned variable that has not already been tried with both values and undoing the decisions for variables assigned later (and which have been tried with both truth values). If eventually the algorithm discovers that both values of the first assigned variable always lead to a conflict, the formula is declared unsatisfiable.

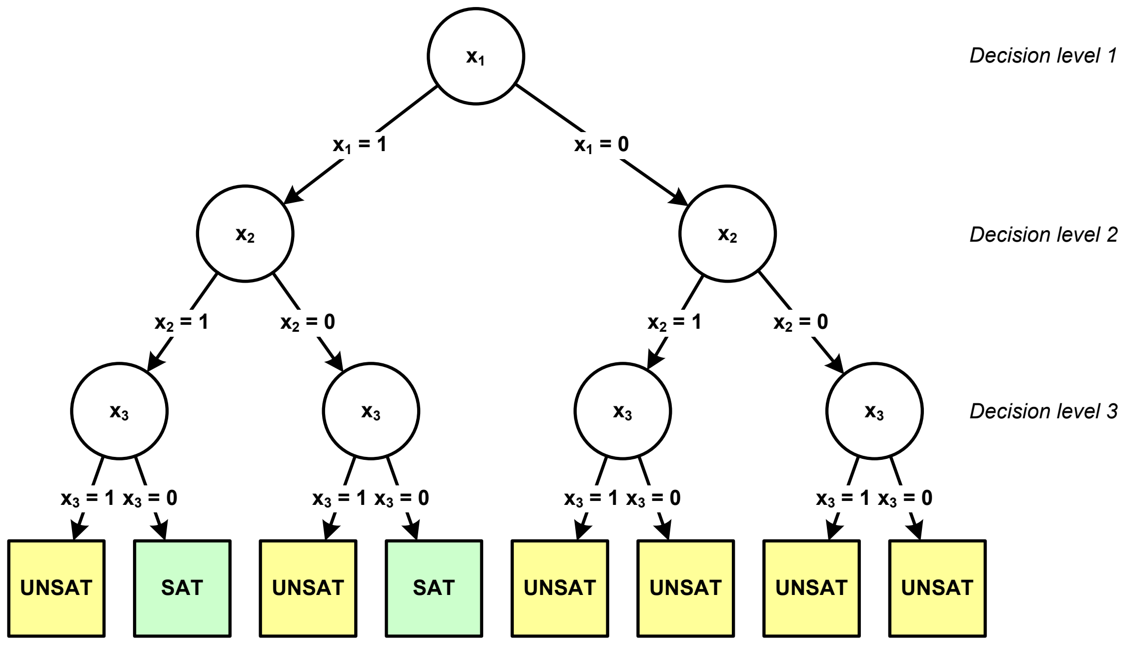


Fig. 3. Search tree example for formula $(x_1 \vee \neg x_2) \wedge (\neg x_3) \wedge (x_1 \vee x_3)$.

The DPLL algorithm can be seen as a depth-first search in the binary tree of partial variable assignments, which is commonly called the search tree. Each intermediate vertex of the tree is associated with a variable and the outgoing edges from that vertex to its offspring represent the possible decisions to assign values true (1) or false (0) to that variable. A path in the tree from the root to any vertex represents a partial assignment to the variables associated with the vertices in the path, so that each variable is assigned with the value corresponding to the outgoing edge taken in the path. A leaf of the tree is a vertex of two possible kinds: SAT or UNSAT. A path representing a full satisfying assignment ends with SAT leaf; a path representing an unsatisfying assignment (either full or partial) ends with an UNSAT leaf.

Fig. 3 shows an example of a search tree for formula $(x_1 \vee \neg x_2) \wedge (\neg x_3) \wedge (x_1 \vee x_3)$ with the decision order being such that x_1 is assigned first, then x_2 and lastly x_3. With this definition in place, DPLL algorithm can be viewed as a depth-first search in the search-tree for a path leading to a SAT vertex. In the example on Fig. 3 there are two such paths: one corresponding to the assignment $\{x_1 = 1, x_2 = 1, x_3 = 0\}$ and another corresponding to the assignment $\{x_1 = 1, x_2 = 0, x_3 = 0\}$. The depth of the tree is the same as the number of variables in the formula being solved. The number of leaves in the tree is the same as the number of full variable assignments, which is exponential in the number of variables in the formula. Decisions for variables at depth *i* in the tree are said to be on *i*'th decision level. Thus, x_1 is said to be assigned on the first decision level, x_2 on the second decision level, and x_3 on the third decision level.

Noticeably, the DPLL algorithm does not necessarily need to traverse the whole search-tree to find a solution. Whenever a partial assignment is known to falsify the formula the algorithm backtracks, so that the subtree of that partial assignment is left unexplored, since it is known to contain only UNSAT leaves. As a result of backtracking the search is directed into another branch of search-tree. For example, assuming the same formula $(x_1 \vee \neg x_2) \wedge (\neg x_3) \wedge (x_1 \vee x_3)$ and a decision order $x_1 \rightarrow x_3 \rightarrow x_2$ that first tries the value 1 for the variables, Fig. 4 shows the search tree where the dashed part is not actually explored by the algorithm, since it is known to lead to UNSAT answers.

Fig. 5 shows the general algorithm framework of an iterative implementation of DPLL algorithm. Modern DPLL-based solvers differ in the methods and the implementation of the following aspects of the general DPLL algorithm:

- decision heuristics implemented by Decide() (a.k.a. branching heuristics): the method by which the choice of the next variable to branch on is made;
- methods for deduction of necessary variable assignments, when given a partial assignment to the variables, such as Boolean Constraint Propagation implemented by BCP();
- methods for conflict resolution and backtracking, implemented by ResolveConflict();
- the underlying data structure for representation of the clause set;
- preprocessing techniques and other optimizations.

The following sections briefly overview some of these aspects.

3.3. Boolean constraint propagation

Boolean constraint propagation (BCP) is an optimization technique used in the basic DPLL algorithm that is a particular case of a family of optimizations commonly called *deduction of necessary assignments*. Necessary assignments to yet unassigned variables

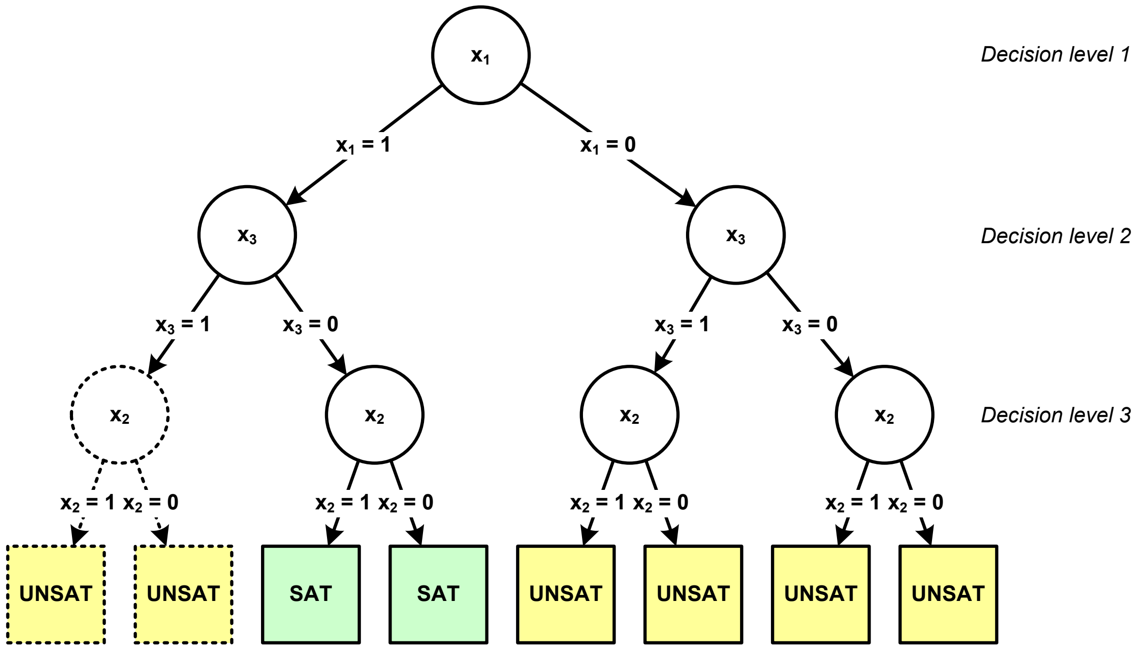


Fig. 4. Example of the search tree for formula $(x_1 \vee \neg x_2) \wedge (\neg x_3) \wedge (x_1 \vee x_3)$, where the dashed part is left unexplored after the assignment $\{x_1 = 1, x_3 = 1\}$.

are those assignments that *must* be done in order to satisfy the formula, according to the current partial variable assignment. BCP is based on the *unit clause rule* that is one of the most efficient rules by which necessary assignments can be identified. The unit clause rule states that if for a certain clause all but one of its literals have been assigned the value false, then the remaining unassigned literal must be assigned with value true in order not to falsify the clause and, consequently, the whole formula. Such clauses are called *unit clauses* and the unassigned literal is called a *unit literal*. BCP, also known as *unit propagation*, is the process by which unit literals are identified and assigned. Note that during BCP an assignment to one unit literal can cause more clauses to become unit and thus result in additional necessary assignments. The BCP process is, therefore, an iterative application of the unit literal rule until no more unit literals exist.

Variable assignments performed during the BCP process are called *implications* and the assigned variables – *implication variables*, to distinguish them from the assignments performed by the algorithm as decisions. Implication variables are considered assigned on the same decision level as the decision variable, the assignment to which triggered the corresponding implications. When a decision is undone during backtracking, the implication variables on the corresponding decision level are unassigned together with the decision variable.

Continuing with the previous example formula $(x_1 \vee \neg x_2) \wedge (\neg x_3) \wedge (x_1 \vee x_3)$, if the algorithm first assigns x_3 with value false, then the clause $(x_1 \vee x_3)$ becomes a unit clause. The BCP process will then make an implication about x_1 having value true. This way the exploration of the search-tree under the branch $\{x_3 = 0, x_1 = 0\}$ is avoided, since it is known to lead to contradiction.

A conflict is an attempt to assign a variable with two different values simultaneously. Without BCP, a conflict can occur when the algorithm makes a decision causing a clause to have all its literals have the value false. In DPLL algorithm with BCP, conflicts can arise only during the BCP process, since decisions are made only for variables that have no unit literals.

3.4. Decision heuristics

The approach by which the algorithm selects variables to assign with a value is commonly called *decision heuristics*, or *branching heuristics*. A number of techniques exist for this purpose, starting from random selection of variables and ending with dynamically changing strategies based on complex statistical analysis of the formula. An overview of various decision heuristics and their impact on the performance of the solvers is presented in [51].

A decision heuristic is considered good if (i) it is computationally simple and imposes little overhead on the algorithm performance; (ii) it allows for quick finding of a satisfying assignment if there is such, and quick finding of contradictions when there is none such.

Most branching heuristics are based on a statistical analysis of the formula. Bohm's heuristic [29] and the Maximum Occurrences in Minimum Sized Clauses [30] methods have proven themselves efficient for some classes of random problems. It has been empirically shown, however, that on non-random SAT instances these heuristics do not behave well enough.

```
while (true)
{
      if (! Decide())
      {
            return SATISFIABLE; // No more unassigned variables
      }
      while (! BCP())
      {
            if (! ResolveConflict()) // A conflict occurred during BCP
            {
                  return UNSATISFIABLE;
            }
      }
}
```

Fig. 5. Iterative implementation of DPLL algorithm (in C syntax).

Benchmarking SAT solvers on non-random test benches has shown that most successful branching heuristics are those that impose minimal overhead on the run-time. For Chaff SAT solver [23] the authors proposed heuristics called Variable State Independent Decaying Sum (VSIDS), which appeared to be very competitive and imposed practically no overhead on the solution run-time. This scheme keeps a score for each literal, which is the number of occurrences of the literal in the formula, and updates it as new clauses are added due to conflict-driven learning (described later). The scheme chooses the variable with the highest score as the variable to branch on. Periodically all the scores are divided by a constant number, thus giving priority to literals appearing in the most recently generated conflict clauses. It is therefore a dynamic strategy, as it adjusts itself to the changes in the formula to give preference to the most recently discovered information.

In Berkmin solver [31] the designers further improved VSIDS scheme. In VSIDS the “activity” of a variable is measured by the number of its occurrences, while in Berkmin the activity is measured by occurrences of the variable in conflicts – at the moment a conflict is discovered by the algorithm, the variables participating in it get their scores increased. In VSIDS the focus on “recent activity” of a variable is based only on the periodic decaying of the scores; in Berkmin the scores are also periodically decayed, but in addition the branch decision is limited to the variable participating in the last added clause that is unresolved.

Developing good decision heuristics is an active research area, and other decision heuristics have been tried out in various works, e.g. [55]. Besides general-purpose approaches, special-purpose heuristics also exist. For example, [56] describes several strategies that are useful in solution of SAT instances encoding BMC problems.

3.5. Deduction of necessary assignments

Deduction of necessary assignments is a technique to find out at a particular moment those assignments that *must* be done to satisfy the formula, according to the current partial variable assignment. One of the most important techniques in this area is BCP, which was described above, which proved itself critical for good solver performance.

Other deduction rules are used in SAT solvers, many of which have been shown to work on specific classes of formulas, but none of which has been shown to be as efficient as the unit clause rule. On general SAT instances most of the other rules only affect the run-time for the worse.

One of the other widely known rules is the *pure literal rule*: if a variable only occurs in a single phase (0 or 1) in all the unresolved clauses, then it can be assigned with a value making that phase evaluate to true. However, detection that a variable satisfies the pure literal rules has been found to be a rather expensive operation during the solution process; thus it is believed that the pure literal rule generally slows down the process for most of the benchmarks.

3.6. Conflict analysis

The situation when a contradiction is found by DPLL algorithm is called a *conflict*. As mentioned earlier, conflicts may arise during the BCP process, when two opposite implications about a variable are made. When this happens, there is always a unit clause that makes an implication that contradicts an already existing variable assignment. This clause is called the *conflicting clause*. The conflicting clause gets all its literals assigned with value false, if the implied assignment is not done, and thus falsifies the whole formula. Therefore, whenever a conflicting clause is found, there is no need for the solver to continue with the current partial assignment, since it is guaranteed that the formula will not be satisfied under it.

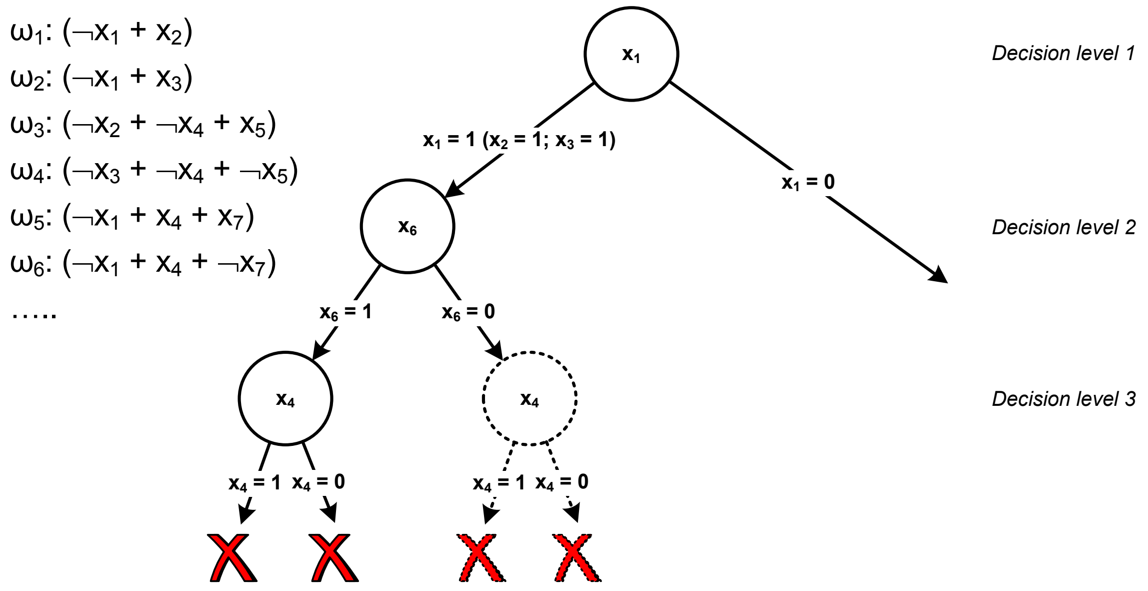


Fig. 6. Example of conflicts for which only the decision $\{x_1 = 1\}$ is responsible. The exploration of the dashed part of the search tree can be avoided by analyzing the reason for the first two conflicts.

When a conflict happens it needs to be resolved, so that the algorithm recovers from the unsuccessfully made decisions. *Conflict resolution* is the general term to describe a technique that is used to recover from conflicts and direct the algorithm out of the search space where there is no solution to another one. Conflict resolution techniques differ by how conflicts are analyzed, how backtracking is performed, and how information is learned to prevent similar conflicts to happen again.

The simplest method of conflict resolution is to undo all the latest assignments up to the latest decision on a variable that has not been tried with both truth values, assign that variable with the value that has not been tried, and continue the search from there. This method is called *chronological backtracking*, since it always tries to undo the last decision that has not been tried both ways. Though on random SAT instances chronological backtracking has been shown to behave well, on structured problems it is generally not efficient.

A more sophisticated conflict analysis aims at analyzing the real reason for the conflict. The benefit of analyzing the real reason for a conflict arises from the fact that not all the decisions made by the algorithm at the moment the conflict happened actually

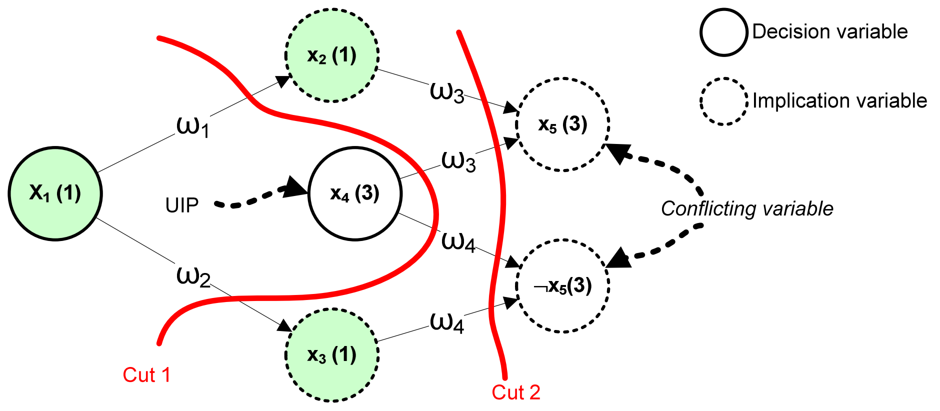


Fig. 7. Example of an implication graph resembling the solution process shown in Fig. 6 that led to a conflict on x_5.

affected the appearance of the conflict. Knowledge of the reason for a conflict may allow a more optimal backtracking and prevention of the same conflict from happening again. For example, in Fig. 6 a conflict arises on variable x_5 after the decisions $\{x_1 = 1, \ x_6 = 1, x_4 = 1\}$. A chronological backtracking scheme would cause the algorithm to explore the branch $\{x_1 = 1, x_6 = 1, x_4 = 0\}$, just to discover another conflict on variable x_7. The only assignment responsible to both these conflicts is the assignment $\{x_1 = 1\}$ on the decision level 1. Despite this fact the chronological backtracking scheme would explore the branch $\{x_1 = 1, x_6 = 0\}$, even though it is known to contain no solution. By finding out the real reason for the conflict, the algorithm may backtrack to an earlier decision level. This method is called *non-chronological backtracking* and it aims at preventing decisions that will generate a sequence of conflicts actually following from the same reason. Continuing the example in Fig. 6, a non-chronological backtracking scheme would backtrack to decision level 1, thus redirecting the search into the branch $\{x_1 = 0\}$.

Conflict analysis can be formulated as a resolution process between clauses involved in BCP process that led to the conflict. However, most often conflict analysis is presented as an analysis of a so-called *implication graph* [27]. Whenever a conflict occurs, the implication graph can be used to model the BCP process that resulted in the conflict. Vertices of the implication graph represent variable assignments and edges represents clauses. An outgoing edge from a vertex v_1 means that the corresponding clause became unit because of the variable assignment represented by v_1. An incoming edge into a vertex v_2 means that the variable assignment represented by v_2 was implied by the corresponding clause when it became a unit clause; such a clause is called the *antecedent clause* of the assignment represented by v_2. A conflict is realized in the implication graph by opposite assignments to the same variable. Continuing the example in Fig. 6, the implication graph resembling the BCP process that led to the conflict on x_5 is shown in Fig. 7. The number in parentheses in each vertex specifies the decision level on which the assignment represented by the vertex was made. White vertices represent assignments made on the current decision level (when the conflict occurred); shaded vertices represent assignments made on earlier decision levels. Although there are more assignments on earlier decision levels, only those that are connected to the white vertices are shown, as they are the only ones that possibly caused the conflict.

Given two vertices v_1 and v_2 on the current decision level in an implication graph, v_1 is said to *dominate* v_2, if and only if any path from the decision variable to v_2 goes through v_1. A *unique implication point* (UIP) is a vertex on the current decision level that dominates both vertices representing the opposite assignments to the conflicting variable. For a given conflict there may be more than one UIP; and the decision variable is always a UIP. Intuitively, any UIP can be seen as a sole reason for the conflict on the current decision level.

Analysis of a conflict with an implication graph involves bi-partition (or a cut) of the implication graph, so that all the decision variables are on one side (the reason side) and the conflicting variable is on the other side (the conflict side). All vertices on the reason side that have outgoing edges going to the conflict side comprise a "reason" for the conflict. This reason can be formulated as a clause that consists of the negations of the literals from which the edges crossing the cut outcome. Such a clause is called a *conflict clause* and it specifies that the assignments comprising the reason for the conflict cannot occur together, i.e. at least one of them must be opposite to what it currently is. Different analysis schemes correspond to different ways the bi-partition of the implication graph is performed, and choice of a specific scheme is heuristic. For example, cut 1 on Fig. 7 constitutes a reason for the conflict on variable x_5 by the fact that both x_1 and x_4 have the value true. A conflict clause formulating this reason and preventing the same conflict from happening again is $(\neg x_1 \vee \neg x_4)$. For cut 2, on the other hand, the conflict clause would be $(\neg x_2 \vee \neg x_3 \vee \neg x_4)$.

The conflict clause produced during conflict analysis as described above can be used to implement non-chronological backtracking: the algorithm can safely backtrack to one level above the maximal decision level of the variables in the produced conflict clause except the variable on the current decision level. That decision level is the lowest decision level that is known to necessarily lead to the conflict.

It has been empirically shown that the heuristic that usually outperforms others is the 1UIP heuristic, which bi-partitions the graph in a way that the UIP closest to the conflicting variable (the first UIP) resides on the reason side of the graph and the assignments implied by this UIP reside on the conflict side. Conflict clauses produced by bi-partition schemes based on UIPs possess an important characteristic: when non-chronological backtracking is performed as described above, then after the backtracking the conflict clause will be a unit clause. Such a clause is called an *asserting clause*, and the only unassigned variable within it is called an *asserting variable*. Having an asserting conflict clause simplifies the decision how to continue the search after the backtracking.

The conflict clauses produced as a result of conflict analysis are redundant in nature, because they are implied by the formula being solved. Still, they may be recorded and added to the formula by the process which is accordingly called *conflict clause recording*, or *learning*. The recorded clauses, though redundant, can often help prune the search space in the future, since they record a reason for a conflict compactly. Thus, for example, when a conflict arises on variable x_5 in the example on Fig. 7 after the decisions $\{x_1 = 1, x_6 = 1, x_4 = 1\}$, the reason for the conflict can be recorded as an additional clause $(\neg x_1 \vee \neg x_4)$, which will prevent the assignment $\{x_1 = 1, x_4 = 1\}$ from happening again in the future.

Conflict-driven learning – the combination of conflict analysis, conflict clause recording and non-chronological backtracking, – has been incorporated into practically all modern SAT solvers, since it proved itself extremely effective in pruning the search space and therefore speeding up the search.

During the solution process a large number of learned clauses may be generated, and storing all of them may slow down the solution process. Therefore, it is often useful for a solver not to store or to delete some of the less necessary learned clauses. There are various heuristics to measure the usefulness and relevance of the learned clauses based on the literal count, the age of the clause, etc. For example, Berkmin [31] measures the usefulness of a particular clause by the number of conflicts in which that clause was involved.

3.7. Formula representation data structures

The data structures used by a SAT solver to represent the formula being solved have critical influence on the performance of the solver. In [32] the authors present a short overview of efficient data-structures for representation of SAT problems, as they are used in the modern state-of-the-art solvers. It has been empirically shown that BCP and backtracking account for the most significant parts of the run-time, therefore efficient data structures aim at minimizing the time required for literal assignment, literal unassignment, and finding unit and unsatisfied clauses after each variable assignment.

One of the most efficient formula representations is the Two Watched Literals scheme [23]. In this scheme the formula is represented as a list of clauses and a clause is a list of literals. Additionally, each clause constantly tracks its two arbitrary distinct literals that are not assigned to false (the watched literals). A clause that has less than two literals not assigned to false is either a unit clause (if there is one such literal) or a conflicting one (if there are no such literals at all). Every variable keeps track of clauses in which its literals are watched. Whenever a variable is assigned a value, either its positive or negative literal becomes false. Apparently, only the clauses that watch the literal that became false may potentially become unit after the assignment and therefore need to be checked. The check is performed by scanning the clause in search for another literal not assigned false. If such a literal is found, then the clause did not become unit and the found literal will now be watched instead of the assigned one. If such a literal is not found, then the clause either became unit or conflicting and appropriate actions can be taken by the algorithm.

Whenever a variable is unassigned during backtracking, no clause should be updated. Clauses that do not watch the literals being unassigned watch other two literals that are not assigned false. Clauses that watch the literals being unassigned, and which previously watched less than two unassigned literals, now become to watch one more unassigned literal, and therefore do not require updating. Note that unassignments happen in reverse chronological order than assignments, therefore it is guaranteed that the last assigned (and therefore last watched) literals will be the first to get unassigned (and therefore become watched).

The Two Watched Literals scheme appears very efficient, because of the low overhead to detect unit and conflicting clauses during variable assignment and no overhead for unassignment.

3.8. Restarts

Restarts are a technique to speed-up the search, which stops the search at some point and restarts it from the beginning by unassigning all variables, thus bringing the search into a new space. This technique may prevent the algorithm from staying in a dead-end of the search tree for a long time and has proved to be efficient on real-life instances. Noticeably, the learned clauses may be preserved during restarts.

Restarts can harm the completeness of a search algorithm, if no technique is used to ensure that the whole search tree is eventually visited. One such technique is to increase periodically the time intervals between restarts, so that eventually the algorithm gets a chance to visit the whole search tree without restarting.

3.9. Preprocessing

Preprocessing techniques are methods applied to formulas prior to the main search algorithm. These techniques aim at simplifying the formula to enable faster completion of the search later. An overview and comparison of various preprocessing techniques can be found in [33]. Such techniques include, among others, symmetry breaking, deduction of necessary assignment, addition of implied clauses, deletion of redundant clauses, identification of equivalent literals; they are out of scope of this work.

Chapter 4

QBF decision procedures

During recent years Quantified Boolean Formula (QBF) solvers have attempted to follow in the footsteps of the dramatic success of propositional SAT solvers. In 2003 the first competition of QBF solvers took place as part of the International Conference on Theory and Applications of Satisfiability Testing (SAT), 2003 conference. Eleven solvers participated in the evaluation, and the results of the evaluation are found in [26].

Most of the QBF solvers implemented in recent years extend the DPLL procedure for SAT. In fact the propositional SAT problem is a restricted case of the QBF validity problem, where all variables can be considered existentially quantified. The following sections describe the techniques used by the extended DPLL procedure for QBF.

4.1. Extension of DPLL for QBF

Satisfiability of a propositional formula *f* can be formulated as the problem of validity of a quantified formula $f' = \exists x_i : f$, where all the variables of *f* are existentially quantified. To determine the validity of f' it is sufficient to find one satisfying assignment to the variables, and hence propositional SAT solvers stop the search as soon as one satisfying assignment is found. In the presence of universally quantified variables it is generally insufficient to find one satisfying assignment, giving particular values to existential and universal variables, to prove validity. Such an assignment does not

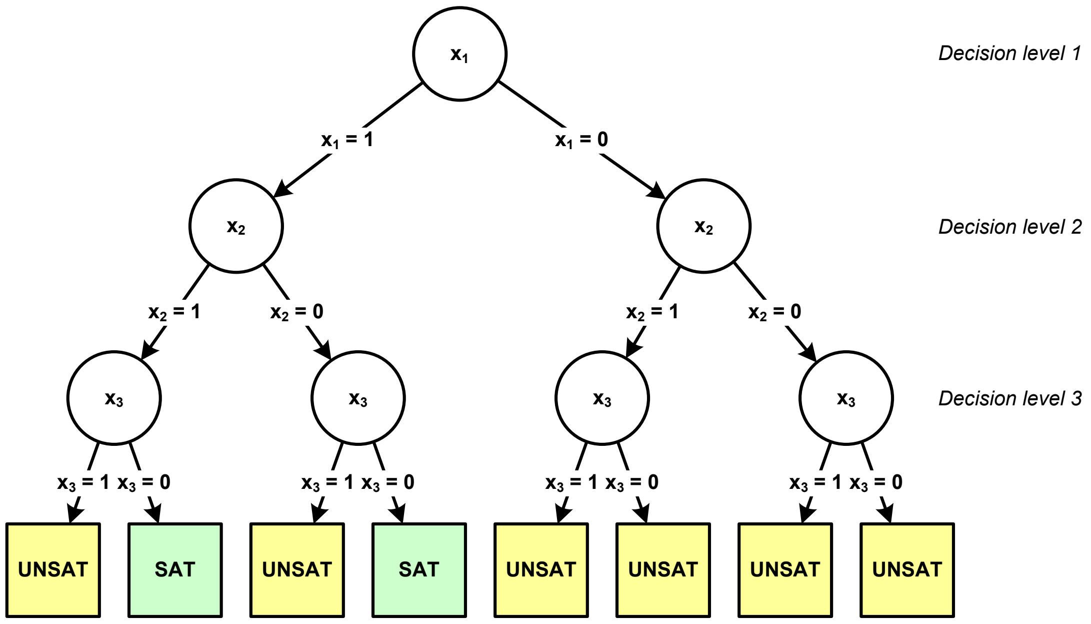


Fig. 8. Example of a search tree for formula $\exists x_1 \forall x_2 \exists x_3 . (x_1 \vee \neg x_2) \wedge (\neg x_3) \wedge (x_1 \vee x_3)$.

guarantee that the formula will be satisfied with other values of the universally quantified variables. This fact lies at the base of the main algorithmic difference between SAT and QBF problems: a SAT solver looks for *one* satisfying assignment and stops as soon as such is found, while a QBF solver should enumerate many satisfying assignments, trying out all possible values of universally quantified variables.

Similary to propositional satisfiability, the search space of the DPLL extension for QBF is a tree of partial variable assignments. An example of such a tree is shown in Fig. 8 for the formula $\exists x_1 \forall x_2 \exists x_3 . (x_1 \vee \neg x_2) \wedge (\neg x_3) \wedge (x_1 \vee x_3)$. To prove validity of this formula the algorithm must visit both SAT vertices of the search tree, thus making sure that for any value of x_2 there is an assignment for x_3 that satisfies the formula.

Noticeably, as in the case of DPLL for SAT, there are useful pruning techniques that allow one to avoid the exploration of those parts of the search tree where the result of the search is known. In the case of DPLL for SAT, such techniques are used to avoid exploration of the search space where no solution is known to exist; however in the case of DPLL for QBF, additional techniques are used to avoid exploration of the search space where a solution is known to exist.

```
while (true)
{
      if (! Decide())
      {
            // A satisfying assignment found

            if (! AnalyzeSAT())
            {
                  // No more values to try for universal variables
                  return VALID;
            }
      }

      while (! BCP())
      {
            // A conflict occurred during BCP

            if (! ResolveConflict())
            {
                  return INVALID;
            }
      }
}
```

Fig. 9. Iterative implementation of the extended DPLL for QBF (in C syntax).

Fig. 9 shows the extended DPLL algorithm framework for QBF. The main difference from the original algorithm shown in Fig. 5 is that whenever a satisfying assignment is found the search does not stop. The formula is reported valid only when no more values remain to try for the universal variables.

The implementation of separate steps of the algorithm is discussed in the following sections.

4.2. Formula representation

As in DPLL for SAT, its extension to QBF operates on a CNF representation of formulas, namely an expression of the form $Q_1x_1Q_2x_2\ldots Q_nx_n{:}\varphi$, where every Q_i is a quantifier, either existential $\exists$ or universal $\forall$, x_i are distinct variables, and φ is a propositional formula over the variables $x_1\ldots x_n$ in CNF, as described for the propositional case.

Noticeably, if x_i and x_{i+1} are equivalently quantified, then the order between them is not important and can be changed without affecting the validity of the formula.

Therefore, the quantification prefix $Q_1x_1Q_2x_2...Q_nx_n$ is often represented as an ordered sequence of variable sets, each set quantified differently than its predecessor. If the last quantification Q_nx_n in the quantification prefix is a universal one, then it is redundant and can be safely removed by eliminating from φ all literals of x_n. Therefore, without loss of generality, it is further assumed that the last quantification is always existential.

The CNF representation enables efficient BCP, conflict detection and conflict-driven learning in DPLL for both SAT and QBF problems. For QBF problems, where the algorithm does not stop on the first encountered satisfying assignment, found solutions can be used to derive information useful for pruning the search process in the future. This *satisfiability-driven learning* is described later, but for its purpose the CNF representation is augmented with a disjunction of cubes (conjunctions of literals), resulting in a so-called *Augmented CNF* (ACNF). ACNF is a combination of CNF for representing the original and learned clauses and disjunctive normal form (DNF) for representing the learned cubes:

$$C_1 \wedge C_2 \wedge ... \wedge C_n \vee D_1 \vee D_2 \vee ... \vee D_m \tag{10}$$

where C_i are clauses (disjunctions of literals) and D_j are cubes (conjunctions of literals), all of which are implied by $C_1 \wedge C_2 \wedge ... \wedge C_n$ and serve exclusively pruning purposes. CNF is a special case of ACNF, therefore whatever is explained in relation to ACNF in the following text applies to the CNF representation as well.

4.3. Boolean Constraint Propagation

In DPLL for SAT one of the primary workhorses of the algorithm is the BCP process. There are also other techniques for deduction of necessary assignments, as described in section 3.5.

The rules applicable in SAT can be extended to apply and be equally useful for QBF as well. For example, the unit clause rule, on which BCP is based, is extended in the following way [37]:

If the ACNF formula contains a clause *C* with an unassigned existential literal *a*, so that:

- all other existentially quantified literals in *C* have value false,

- all universally quantified literals with lower quantification level have value false, and
- all universally quantified literals with higher quantification level are unassigned,

then the literal *a* must be assigned with value true for the formula to be satisfied.

This unit clause rule refers only to existentially quantified literals (in SAT all literals are existentially quantified). For QBF in ACNF form, a dual rule can be formulated for universally quantified literals and is called a *unit cube rule* [37]:

If the ACNF formula contains a cube *S* with an unassigned universally quantified literal *x*, so that:

- all other universally quantified literals in *S* have value true,
- all existentially quantified literals with lower quantification level have value true, and
- all existentially quantified literals with higher quantification level are unassigned,

then the formula is satisfied unless we assign false to *x*. As the unit clause rule is used to avoid conflicts and save the exploration of the search space known to contain no solution, the unit cube rule can be used to avoid exploration of the search space known to surely contain a solution by assigning *x* with False.

The BCP process for QBF in ACNF is based on the two rules described above: it is an iterative application of unit clause and unit cube rules until no more implications may be deduced.

To illustrate the effect of BCP based on unit cube rule consider the following valid QBF:

$$\begin{aligned}
&\forall x_1 \exists x_2 \forall x_3 \forall x_4 \exists x_5 : C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \\
&\textit{where} \\
&C_1 = (x_1 \vee \neg x_3 \vee x_5) \\
&C_2 = (\neg x_1 \vee x_2 \vee x_5) \\
&C_3 = (\neg x_2 \vee x_4 \vee x_5) \\
&C_4 = (x_3 \vee \neg x_4 \vee \neg x_5) \\
&C_5 = (x_2 \vee \neg x_3 \vee \neg x_5)
\end{aligned} \tag{11}$$

Note that whenever $\{x_2 = 1; x_3 = 0\}$ the CNF is satisfiable, and therefore the cube $D_1 = (x_2 \wedge \neg x_3)$ is implied by this formula and can be safely added to it, transforming the formula to the following ACNF:

$$\forall x_1 \exists x_2 \forall x_3 \forall x_4 \exists x_5 : \left(C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5\right) \vee D_1 \qquad (12)$$

Now assume that at some point x_2 is assigned value true. The BCP process will immediately assign true to x_3 in order to force the cube D_1 to have value false to avoid the exploration of the search space obviously having a solution.

4.4. Decision heuristics

As in the case of propositional SAT, a good choice of decision heuristics is critical for the speed of the algorithm. In DPLL for SAT there is no restriction on the order in which the algorithm chooses variables to split upon, since all the variables belong to the same (and the only) existentially quantified set. In DPLL for QBF this is not the case, and the choice of the next variable to branch on is subject to the following conditions:

- Variables from the same quantified set can be chosen in any order, because $\forall x \forall y : \phi = \forall y \forall x : \phi$ and $\exists x \exists y : \phi = \exists y \exists x : \phi$.
- Variables from different quantified sets should be chosen according to quantifier nesting, from the outermost one to the innermost one, because $\exists x \forall y : \phi \neq \forall y \exists x : \phi$.

Unless special techniques are used, such as inversion of quantifiers (described later), QBF solvers obey this restriction on the decision heuristics. For the choice of variables from the same quantified set, QBF solvers usually use the same kinds of heuristics as propositional SAT solvers, e.g. VSIDS.

The restriction on the order of decisions is a fundamental difference between SAT and QBF, and is considered to be one of the key limitations of QBF compared to SAT.

4.5. Conflict analysis and satisfaction analysis

In propositional SAT the conflict is said to occur when two contradicting implications are made about a variable, or a clause gets all its literals assigned with the value false. This rule (the conflicting rule) is not directly applicable in QBF. As mentioned in the description of BCP for QBF above, a clause in ACNF may be unit even when more than one of its literals is unassigned: what counts is only the number of existentially quantified literals. Hence, the conflicting rule for QBF is accordingly adjusted, and a conflict is said to occur when two contradicting implications are made about the same variable, or a clause gets all its existentially quantified literals assigned with value false and the universally quantified literals either assigned with value false or unassigned (i.e. not assigned with value true).

Efficient conflict analysis coupled with learning and non-chronological backtracking proved to be extremely effective for propositional SAT solvers, especially on non-random instances. As described earlier, in propositional SAT conflict analysis is performed whenever a conflict occurs, and it results in the following artifacts:

- a new clause representing the reason for the conflict, which is added to the original clause database;
- the computed decision level to which the algorithm backtracks at the end of the analysis.

Conflict-driven learning and non-chronological backtracking has been extended also to QBF [35][36][37] and empirically shown useful. Conflict analysis in QBF is formulated by the concept of long-distance resolution, introduced in [35]. In propositional SAT an iterative application of resolution of the conflicting clause with its antecedent clauses in order to eliminate the implication variables results in a clause representing the reason for the conflict. Similarly, in QBF the conflicting clause can iteratively be resolved with its antecedent clauses to produce a conflict clause. The only difference between SAT and QBF is that the resolvent of two clauses in QBF can be a tautological clause. For example, consider the following fragment of a QBF:

$$...\exists x_1 \exists x_2 \exists x_3 \exists x_4 \forall y_1 ....(x_1 \vee x_2 \vee y_1 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg y_1 \vee x_4)... \qquad (13)$$

Assuming x_3 and x_4 have value false, when a decision on x_1 assigns it value false, the first clause becomes unit, because y_1 has a higher quantification level than x_1 and x_2. Therefore, x_2 is deduced to be true, causing the second clause to become conflicting, according to the conflicting rule for QBF. The resolution of the conflicting clause (the second clause) with the antecedent clause of x_2 (the first clause) results in the new clause $(x_1 \vee y_1 \vee \neg y_1 \vee x_3 \vee x_4)$, which is tautological. Nevertheless, it may be added to the database and be useful for pruning the search space, since it will become unit as soon as x_3 and x_4 get assigned value false. This clause can also be used for non-chronological backtracking in the same way as is done in SAT.

Since in QBF the search does not stop when a satisfying assignment is encountered, an additional technique, namely *satisfaction-driven learning*, has been developed to avoid exploration of the search space already known to have a solution. Similary to the case when a conflict is discovered, it is possible to analyze the solution found, derive a reason for it and prevent the algorithm from later encountering a solution implied by the same reason. The result of satisfaction analysis is a cube, which can be used for non-chronological backtracking after a solution is found and can be added to the ACNF database for pruning of the future search.

The method of generating conflict clauses and satisfying cubes in the process of the search is described in [35][36][37].

4.6. Additional QBF-specific techniques

A number of techniques specific to QBF have been developed in the recent years in an attempt to make the DPLL-based QBF solvers faster. However, in practice, none of the specific techniques has proved itself effective in any manner comparable with BCP or conflict/satisfaction-driven learning. For completeness, some of those techniques are presented in the following paragraphs.

4.6.1. Unfolding

Unfolding is the technique of elimination of universal quantifiers from a QBF based on the following rule:

$$\forall x : f(x) = f(x \mapsto 1) \wedge f(x \mapsto 0) \tag{14}$$

Elimination of all universal quantifiers in fact reduces the QBF problem to SAT. Even elimination of some of the universal quantifiers may result in a significant speed up of the search due to the possibility of making substantially more implications by the unit clause rule, as shown in [38]. The drawback of this technique, however, is that the elimination of a universally quantified variable doubles the formula size, causing the unfolded formula to grow exponentially.

In [38] a technique called implicit unfolding is proposed, in which the formula is not physically unfolded, and therefore does not grow up exponentially, but instead a worst-case exponential algorithm of unit clause propagation analyzes the formula as if it were unfolded. The author of [38] then proposes a number of limitations that may be set upon this unit propagation algorithm, which may limit the run-time to sub-exponential. Still the computational cost of the algorithm is high; the trade off between the overhead of the algorithm and its usefulness is not clear. In practice, most successful QBF solvers to-date do not make use of the implicit unfolding technique.

4.6.2. Inversion of quantifiers

Normally, the decision strategy in the DPLL extension for QBF is restricted to choose variables in the order of their quantification, as mentioned in section 4.4 above. In [39] the author proposes a technique that enables reasoning with variables that are not quantified in the outermost quantification. Noticeably, when given the formula of the form $\exists X \forall Y : \phi$, it is useful to look at the formula $\forall Y \exists X : \phi$, where the quantifier order is reversed. If for some valuation of Y only certain valuations of X are possible, then these valuations of X are the only possible for $\exists X \forall Y : \phi$. This observation leads to the technique called *inversion of quantifiers*, which is used to determine necessary assignments to the variables of X. Given the formula $\exists X \forall Y : \phi$, a random valuation is chosen for the variables of Y; then BCP is performed, resulting in a set of necessary assignments to the variables of X. Any number of valuations for Y may be tried out in an attempt to discover more necessary assignments to the variables of X or even conflicting assignments. In the latter case, the formula may be declared invalid.

In [39] this technique is applied to the formula as a preprocessing step prior to the execution of the main search procedure.

4.6.3. Sampling

Sampling is an additional pruning technique which has also been proposed in [39]. For the formula $\exists X \forall Y : \phi$, whenever a variable from X is assigned, the viability of the assignment is unknown at least until all values of variables from Y are tried one way or another. However, the viability of the assignment to a variable from X can be checked faster by making a random valuation of a small number of variables of Y and performing BCP to check that no conflicts arise. This process is called *sampling* of Y and aims at faster detection of wrong decisions.

Sampling is somewhat similar to the inversion of quantifiers. But unlike the inversion of quantifiers, this technique is applied in [39] after each decision on the value of an existential variable.

Chapter 5

jSAT

Chapter 2 presented the results of the evaluation of general-purpose SAT and QBF solvers for the solution of BMC problems encoded in a classic approach with the following propositional SAT instance (15) and a QBF instance (16), avoiding the memory explosion during successive increase of the BMC bound:

$$R_k(Z_0,Z_k) = \exists Z_1,\ldots,Z_{k-1} : I(Z_0) \wedge F(Z_k) \wedge \bigwedge_{i=0}^{k-1} TR(Z_i,Z_{i+1}) \tag{15}$$

$$\begin{aligned} &R_k(Z_0,Z_k) = \exists Z_1,\ldots,Z_{k-1} : I(Z_0) \wedge F(Z_k) \wedge \\ &\forall U,V : \left(\bigvee_{i=0}^{k-1} (U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1}) \right) \rightarrow TR(U,V) \end{aligned} \tag{16}$$

The conclusion of the evaluation was that the general-purpose QBF solvers are totally out of the running compared to the SAT solvers. This inefficiency of the general-purpose QBF solvers on QBF formulations of BMC problems served as motivation for the development of a special-purpose QBF decision procedure, called jSAT, for these specific kinds of QBF. The scope of the development was limited to formulas of form (16); and the goal for the developed special-purpose solver was to improve the run-time performance of the solution to come as close as possible to SAT-based approaches on the corresponding SAT formulations, and still preserve the memory efficiency enabled by QBF formulations.

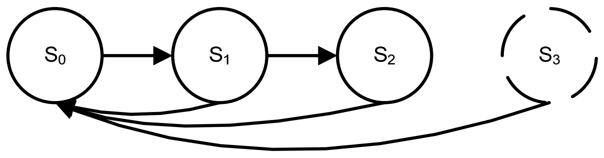


Fig. 10. State graph of a correct model for a resetable 2-bit modulo-3 counter.

5.1. Motivation and approach

The primary reason for the inefficiency of general-purpose QBF solvers compared to SAT solvers is believed to be the lack of efficient Boolean Constraint Propagation between the state encoding variables, caused by the restriction of the decision strategy to assign all state encoding variables before assigning to variables of U and V, as well as by the lack of explicit relations between the state encoding variables in the formula. The relations encoded by $TR(U, V)$ part of the formula do not have any effect until very late in the solution process. Essentially, as mentioned in Chapter 2, QBF solvers first choose all the state encoding variables and only then check whether they constitute a path.

Another reason for the inefficiency of the QBF solvers is the redundant exploration of the search space in which the implication $(U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1}) \rightarrow TR(U,V)$ is trivially satisfied when those valuations of U and V are tried out that do not represent any chosen pair of neighboring states Z_i and Z_{i+1}, i.e. the left side of the implication evaluates to false.

An example of a state graph is shown on Fig. 10, which depicts a correct model for a resetable 2-bit modulo-3 counter, and where S_0 is the initial state and S_3 is the unreachable state where the counter has the illegal value 3. Assume a BMC problem that S_3 is reachable from S_0 in two transitions. A SAT solver on the formula of the form (15) will deduce the values for the encoding variables of Z_0, Z_1 and Z_2 immediately during the first invocation of BCP, completing the solution process without making any decision, because:

- S_0 is the only initial state, whose value is deducible from $I(Z_0)$;

- S_1 is the only successor of S_0, and its value is deducible from $TR(Z_0, Z_1)$ as soon as Z_0 is deduced;
- S_3 is the only final state whose value is deducible from $F(Z_2)$.

The solution process of a QBF solver on the corresponding formula of form (16) is much less efficient. It will also immediately deduce the values for Z_0 and Z_2, but it will perform an exhaustive search for a value of Z_1 that could make the formula valid. Z_1 will be chosen heuristically to be any one of the four states, and then an exhaustive trial of all possible 2^4 values for the encoding variables of U and V will be made to check if all of them satisfy the implication under the universal quantification. Even with learning and non-chronological backtracking, which could help to avoid re-exploration of the already explored search space, the deficiency of the QBF approach is obvious.

The primary goal of the development of jSAT was to overcome, at least partially, the two deficiencies of the general-purpose QBF approach, namely:

- redundant exploration of the search space where the implication $(U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1}) \rightarrow TR(U,V)$ is trivially satisfied; and
- inappropriate choices of neighboring states that do not constitute a path.

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:

$$I(Z_0) \wedge TR(U,V) \wedge F(Z_k) \tag{17}$$

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 (16) with assistance of the terms of the form $(U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1})$, jSAT 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. This way the redundant exploration of the values of U and V that do not correspond to any state is avoided, since U and V always have values making the left side of the implication $(U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1}) \rightarrow TR(U,V)$ evaluate to true.

To avoid choices of states that do not constitute a path, jSAT tries to choose states consistently with the previously chosen ones. For example, when state Z_0 is chosen, and the aliases (U, V) are associated with states (Z_0, Z_1), the relations between Z_0 and Z_1 are explicit in the formula (17), since $TR(U, V)$ in fact represents the relation $TR(Z_0, Z_1)$. The explicit relations between Z_0 and Z_1 allow a consistent choice of Z_1.

Intuitively, the jSAT algorithm can be seen as a bounded depth-first search in the state graph of the system from the initial states to the final ones. The search is bounded because it only goes up to k steps in depth, where k is the bound of the BMC problem being solved.

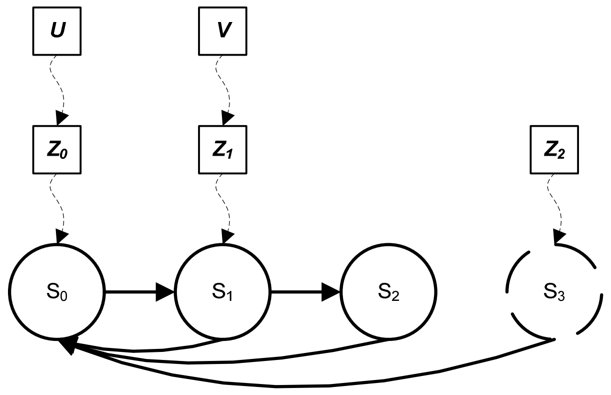
The algorithm starts by associating U with Z_0 and V with Z_1; thus the formula (17) becomes semantically equivalent to:

$$I(Z_0) \wedge TR(Z_0, Z_1) \wedge F(Z_k) \tag{18}$$

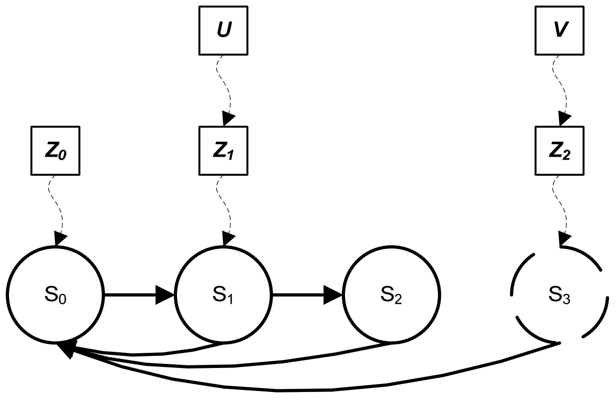
The states Z_0 and Z_1 are then chosen by finding an assignment to their encoding variables, if possible, so that Z_0 is an initial state and Z_1 is its successor. As soon as they are chosen, the algorithm makes Z_1 be the current state and Z_2 be the next one: U becomes an alias of Z_1, and V becomes an alias of Z_2. The algorithm proceeds in this fashion until all states are successfully chosen, or until it discovers that such a choice is impossible. In fact, jSAT solves a sequence of formulas, shown below, having in mind that all the similarly named state encoding variables are equivalent in all the formulas:

$$\begin{gathered} I(Z_0) \wedge TR(Z_0, Z_1) \wedge F(Z_k) \\ I(Z_0) \wedge TR(Z_1, Z_2) \wedge F(Z_k) \\ I(Z_0) \wedge TR(Z_2, Z_3) \wedge F(Z_k) \\ \ldots \\ I(Z_0) \wedge TR(Z_{k-1}, Z_k) \wedge F(Z_k) \end{gathered} \tag{19}$$

Whenever the algorithm fails to choose the next state consistently, the depth-first 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.



(a)



(b)

Fig. 11. jSAT solution process for the 2-bit module-3 counter from Fig. 10. (a) when *U* and *V* are associated with Z_0 and Z_1, values for Z_0, Z_1 and Z_2 are deduced immediately; (b) when *U* and *V* move forward a conflict arises.

Back to the simple example on Fig. 10, Fig. 11a shows that jSAT will deduce Z_0 and Z_2 immediately just like a SAT or a general-purpose QBF solver. Also, since in the beginning the current and the next states *U* and *V* are associated with Z_0 and Z_1, the value of Z_1 will also be immediately deduced. Upon the next step of the depth-first search, which will try to extend the path $Z_0 \rightarrow Z_1$ one more transition, *U* and *V* will become associated with Z_1 and Z_2 respectively, as shown in Fig. 11b. A contradiction will immediately arise, since there is no transition between these two states, which have already been assigned to be S_1 and S_3, respectively.

```
 1: InitializeCurrentAndNextStates();
 2: while (true)
 3: {
 4:     if (! Decide())
 5:     {
 6:         if (AllStatesDecided()) return VALID;
 7:         if (! AdvanceCurrentAndNextStates()) return INVALID;
 8:     }
 9:     while (! BCP())
10:     {
11:         if (! ResolveConflict()) return INVALID;
12:     }
13: }
```

Fig. 12. Pseudo-code of jSAT decision procedure (in C syntax).

The jSAT solution of the BMC problem in the above example was not as efficient as that of a SAT solver, due to the inability to perform BCP across more than one transition at the same time, since there is only one copy of the transition relation available to jSAT. Still, jSAT was able to solve the problem by only applying BCP without making any decision, and thus was much more efficient than a general-purpose QBF solver.

5.2. Algorithm description

jSAT is based on the DPLL algorithm, which was described in the previous chapters. Fig. 12 shows the structure of the algorithm. The main algorithmic difference of jSAT from the classical DPLL is the action taken on lines 6-7 whenever a satisfying assignment is found to the formula with U and V being aliases to a specific pair of states. Since jSAT iteratively solves different formulas (with U and V being aliased to different states of the path), whenever one formula is satisfied the algorithm does not finish, but adjusts U and V to the next pair of states in order to start solving the next formula. This is somewhat similar to the extension of DPLL for QBF, where the algorithm does not finish when a satisfying assignment is found, but backtracks to check other (still unexplored) branches of the universally quantified variables.

The following sections describe the details of the various steps of the algorithm, such as the decision strategy, conflict analysis technique, etc.

5.2.1. Overview

The algorithm first initializes the states *U* and *V* to be associated with Z_0 and Z_1, respectively. With such setting jSAT will start its search process by finding values for these states first.

The procedure `Decide()` 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. The decision strategy has some restrictions, which are covered in the following paragraphs.

`Decide()` returns true if the decision is made successfully. Boolean Constraint Propagation is then performed by the procedure `BCP()`, which returns false in case a conflict arose. If so, `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.

`Decide()` returns false whenever all the encoding variables of the current and the next states have been decided. If at this point all the states of the path 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. `AllStatesDecided()` returns true when all the state encoding variables are assigned and the current and the next states *U* and *V* are positioned on the last pair of states in the path.

If undecided states remain, `AdvanceCurrentAndNextStates()` 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 the $TR(U, V)$ part of the formula. Since the newly discovered information may contradict some of the already made decisions, conflicts may arise during the adjustment operations. The procedure `AdvanceCurrentAndNextStates()` returns false in case a conflict occurred that could not be resolved; in this case the algorithm terminates and the given formula is invalid.

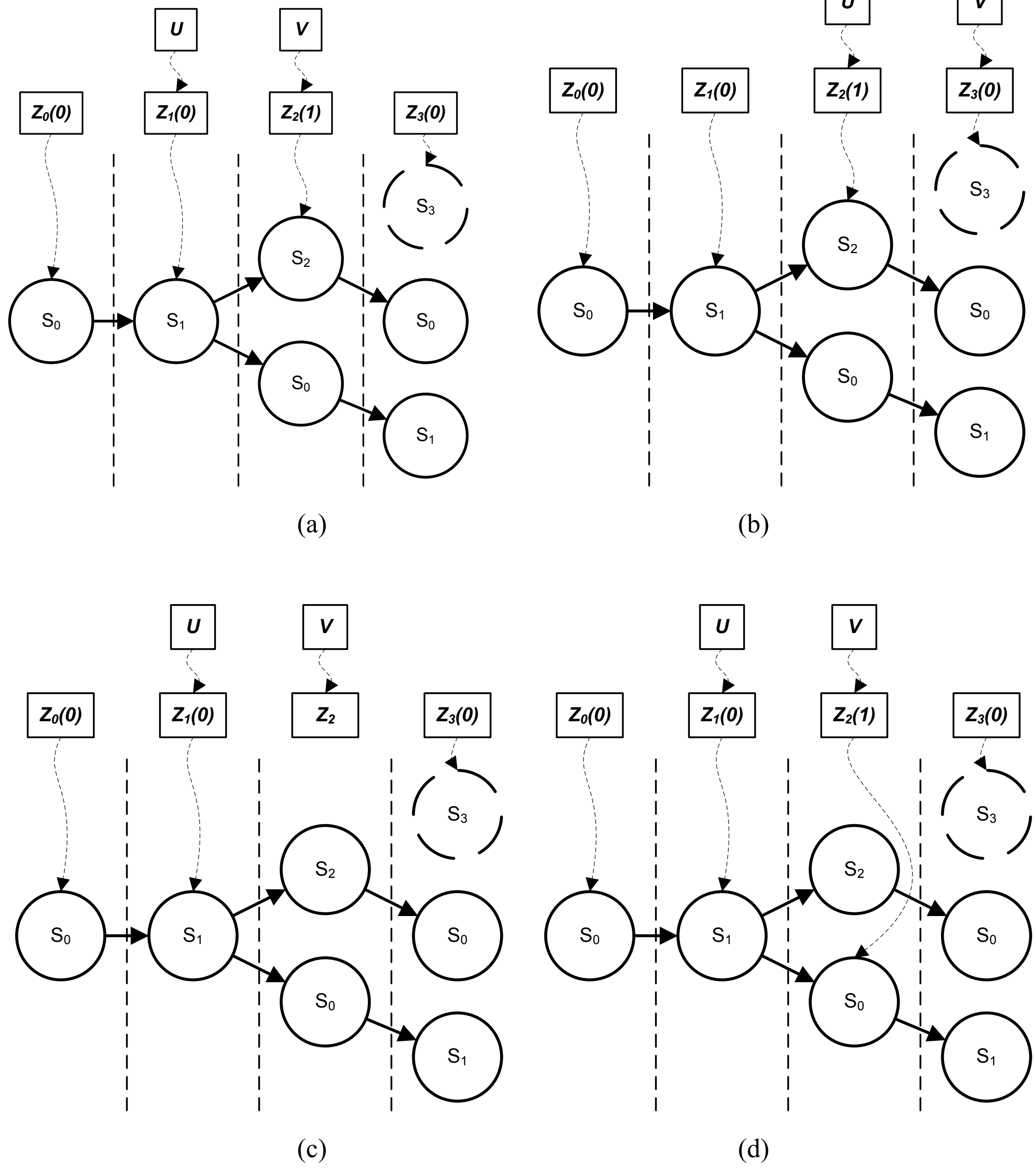


Fig. 13. The jSAT solution process on the example of 2-bit modulo-3 counter from Fig. 10 with BMC bound 3. (a) *U* and *V* are fully assigned, causing the DFS traverse to advance; (b) *U* and *V* after the advancement, showing the conflicting situation; (c) Z_2 is unassigned as a result of backtracking, and *U* and *V* are retracted; (d) as a result of conflict resolution Z_2 is assigned with a new value, redirecting the DFS into a new branch.

5.2.2. Decision heuristics

The decision strategy of jSAT is not as arbitrary as that of the propositional SAT solvers. The following restrictions apply to make jSAT implement a depth-first search of the state graph and to "visit" only the states actually reachable from the initial states:

- Only the encoding variables of the current and the next state are chosen. If there are no more such variables to decide, then the current and the next states should be advanced in the path.
- The encoding variables of the current state are chosen prior to the encoding variables of the next state.

The restrictions described above ensure selection of the 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. Moreover, every state in the path is chosen consistently with the previous state.

The order of the selection of the encoding variables of the same state is not important, and heuristics similar to the ones used in SAT/QBF solvers can be used.

5.2.3. Conflict analysis

Conflicts may arise during the solution process either as a result of BCP performed after a decision is made, or as a result of state adjustment operations (advancement and retreating of the DFS traverse). The purpose of the conflict analysis is, as in the case of other DPLL-based solvers, to find a reason for the conflict, record this reason as a learned clause, and decide on the decision level to which the search should backtrack, thus implementing conflict-driven learning and non-chronological backtracking. Since conflicts in a sense represent a dead end of the DFS traversal in the system state graph, i.e. a situation in which it is known that no path to the final state can be found, there is one more goal for the conflict analysis to achieve in the case of jSAT, namely, identify the state to which the DFS traversal of the system state graph should retreat. There is a close correlation between the decision level to backtrack to (the *backtracking level*) and the retraction of the current and the next states U and V: U and V should be adjusted in such a way that U corresponds to the last fully decided state in the path and V to the state

```
ResolveConflict()
{
    nBacktrackingLevel = AnalyzeConflict();
    if (nBacktrackingLevel < 0)
        return false;

    nFirstUndecidedPathState = Backtrack(nBacktrackingLevel);

    if (! RetractCurrentAndNextStates(nFirstUndecidedState))
        return false;
    return true;
}
```

Fig. 14. Pseudo-code of jSAT conflict resolution procedure (in C syntax).

following it. Thus, the search will proceed by finding another assignment to the variables of *V*, redirecting the search to another branch of the system state graph.

Back to the example of 2-bit modulo-3 counter on Fig. 10, Fig. 13 shows an example of a conflicting situation during solution of BMC problem of bound 3 using an “unrolled” state graph presenting all possible computation paths of length 3. The numbers in parentheses near every state specify the decision level on which the state is completely assigned. The states Z_0, Z_1 and Z_3 are fully assigned on the decision level 0, since there is a single possible initial state Z_0, a single possible transition out of that initial state to Z_1, a single possible final state Z_3. In Fig. 13a the current and the next states *U* and *V* correspond to the pair (Z_1, Z_2), which are fully assigned. At this point the DFS traversal advances by associating *U* and *V* with the next pair of states in the path (Z_2, Z_3), as shown in Fig. 13b. A conflict occurs as a result of this state adjustment operation, because no transition exists from Z_2 to Z_3, which means that the most recently chosen state (i.e. the state encoding variables of which are assigned on the highest decision level) was not chosen correctly. As a result of backtracking, the variables on the highest decision levels are unassigned, resulting with the current state Z_2 associated with *U* to be incompletely assigned. This, in turn, means that another assignment to the encoding variables of Z_2 should be found, still being consistent with the current value of Z_1. To achieve this the DFS traversal retreats by associating the current and the next states *U* and *V* back with the pair (Z_1, Z_2), as shown in Fig. 13c. In the next step another value will be assigned to one of the encoding variables of Z_2, thus redirecting the DFS into another branch of the state graph, as shown in Fig. 13d.

Fig. 14 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 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, as described above. 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.

The procedure `AnalyzeConflict()` uses the same analysis method as other DPLL-based solvers by analyzing the implication graph to find the reason for the conflict. An important aspect of this analysis 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 state encoding variables of the states Z_i. jSAT conflict analysis achieves this by using only decision variables (which are state encoding variables by the restriction on the decision heuristics) in the learned clauses, somewhat similar to the Last UIP learning scheme described in [27].

5.2.4. Formula representation data structures

The CNF representation of the formula (17) on which jSAT operates may be stored in any way that the propositional SAT solvers use. As in the case of propositional SAT solvers, an appropriate data structure for formula representation is crucial for the efficiency of the solution, and thus approaches like the Two Watched Literals scheme are advantageous.

In addition to the CNF representation of the formula (17), jSAT requires knowledge of which are the state encoding variables and what states they encode. Noticeably, the formula (17) does not necessarily contain (and in fact, usually does not contain) references to all the state encoding variables, since the only states explicitly mentioned are the initial and final states. The knowledge of all the state encoding variables is required for the implementation of the decision strategy, the conflict analysis algorithm, and the algorithm of association of the current and the next states U and V with actual states in the path.

5.2.5. Additional optimization techniques

Besides BCP, conflict-driven learning and non-chronological backtracking, other optimization techniques proved themselves useful in case of the propositional SAT and QBF solvers. These techniques, such as periodic restarting for example, may be applied to jSAT as well, but they are out of scope of this work.

5.3. Experimental evaluation

5.3.1. Test generation

Benchmarks for the evaluation of the performance of jSAT versus the propositional SAT solvers and the general-purpose QBF solvers were generated using the method described in Section 2.2.1. jMC model checker was enhanced with an additional engine, the purpose of which was to generate formulas to feed to jSAT solver. Fig. 15 depicts the extended structure of the test generation process and jMC model checker. The formulas for jSAT were generated in JDIMACS format (described in Annex B) , which is a slightly modified DIMACS format that adds the specification of state encoding variables to the file. The same test bench of thirteen model checking examples and the same benchmarking environment as described in Section 2.2.1 were used.

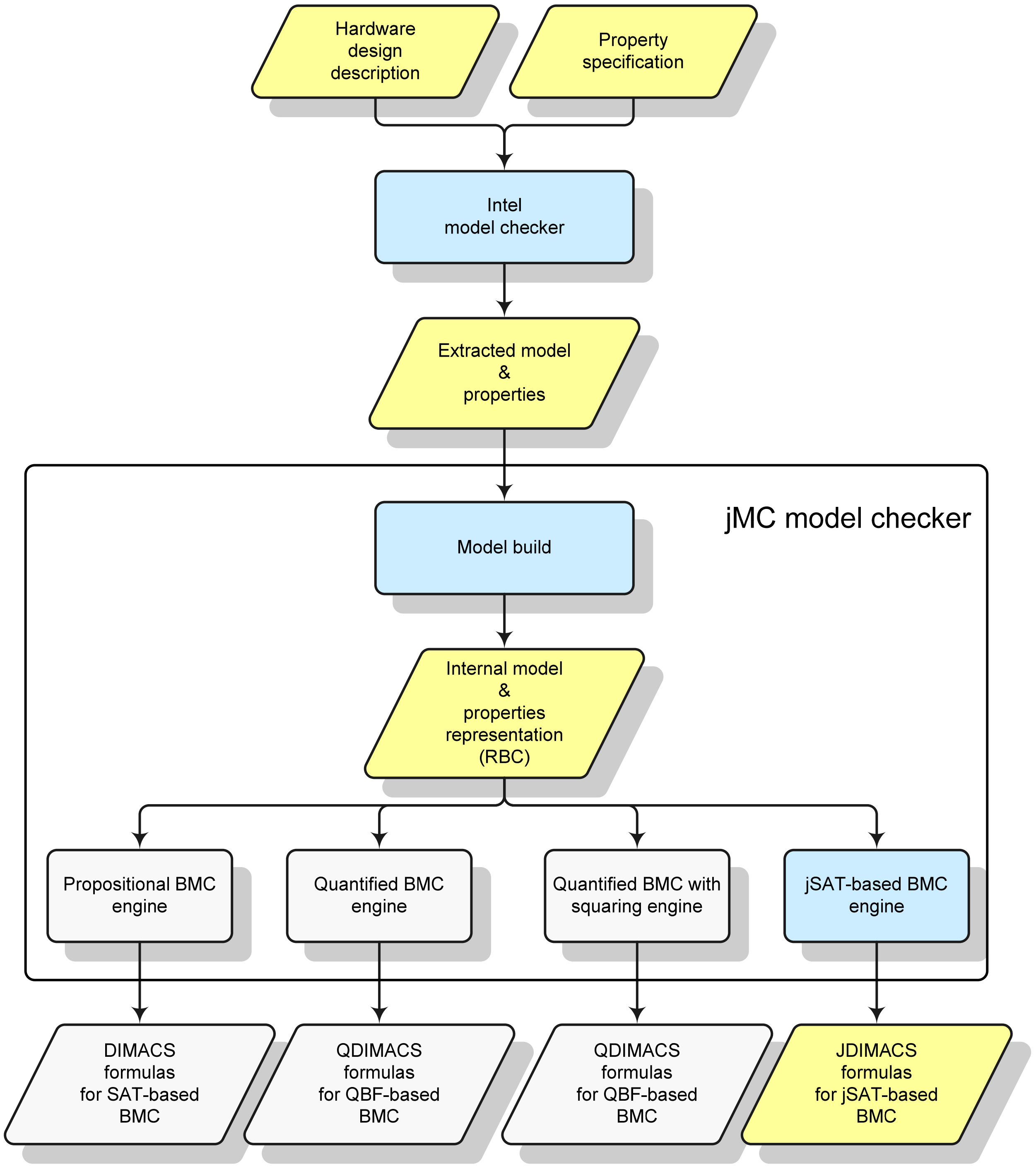

Hardware design description
Property specification
Intel model checker
Extracted model & properties
jMC model checker
Model build
Internal model & properties representation (RBC)
Propositional BMC engine
Quantified BMC engine
Quantified BMC with squaring engine
jSAT-based BMC engine
DIMACS formulas for SAT-based BMC
QDIMACS formulas for QBF-based BMC
QDIMACS formulas for QBF-based BMC
JDIMACS formulas for jSAT-based BMC

Fig. 15. The process of test generation extended to produce tests for jSAT solver.

5.3.2. jSAT implementation

The implementation of jSAT for the purpose of evaluation was performed based on the solver described in [22][45], which was found to be slightly slower than zChaff [23]. The decision to base the implementation on this specific solver followed from its well-designed and clearly documented implementation, which enabled easier implementation of jSAT.

The data structure for the representation of the clause set of the formula being solved was Two Watched Literals. With this data structure the state adjustment operations, used by `AdvanceCurrentAndNextStates()` and `RetractCurrentAndNextStates()`, were implemented with the following simple method:

- unassign all variables, bringing the algorithm to its just-initialized state;
- associate 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; and
- perform BCP.

The decision strategy used in the implementation was a variation of the VSIDS heuristics, subjected to the restrictions, described in Section 5.2.2, where only the state encoding variables are selected for decision and so that earlier states are selected first. Within the encoding variables of the same state a non-dynamic variation of VSIDS was used, i.e. the variable weights were not updated dynamically during the solution process.

The implication graph for the purpose of conflict analysis was not explicitly built, just as it is not built by the base solver [22][45]. Instead, when the BCP process assigns a variable because of a clause becoming unit, that variable is associated with the antecedent clause that implied it. Generally, in the case of the propositional SAT solver using this method, the implication graph can be reconstructed by walking from a variable to the variables participating in its antecedent clause. However, in jSAT, 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. The disassociated variables lose their relations to the corresponding antecedent clauses,

	# 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	

Table 3. Number of instances solved by jSAT and the two SAT solvers per test case. There are 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. The '-' sign specifies that there are no instances with the specific result for the corresponding test case.

resulting with an information loss incurred by the state adjustment operation. In the case of propositional SAT solvers, variables without an associated antecedent clause may only be decision variables. In the evaluated implementation of jSAT they may also be implied variables that lost their relation to the corresponding antecedent clauses. For the purpose of conflict analysis, such variables should be treated as decision variables and included in the learned conflict clauses. Specifically, the produced conflict clauses include not only the decision variables on the decision levels of variables participating in the conflicting clause, but also state encoding variables on those levels that are not decision variables but do not have an associated antecedent clause.

The evaluated jSAT implementation included conflict-driven learning and non-chronological backtracking, but no other optimization techniques, such as restarting.

5.3.3. Benchmarking results

For fair analysis the performance of jSAT was compared against that of [22][45], but the comparison to zChaff is brought for completeness. The summary of the evaluation results is presented in Table 3. The numbers of solved SAT and UNSAT instances are shown separately.

Interestingly, jSAT results are especially close to those of the base solver [22] on SAT instances, where jSAT managed to solve 104 versus 106 instances solved by [22]. On UNSAT instances, the distance between jSAT and [22] is much more significant. Fig. 16 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 solvers. It still did not achieve the same run-times as the propositional SAT solvers, though in the biggest test case, test01 (see Table 1), it managed to solve all the instances in seconds, 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 more complex instances: the slope of the performance curve of jSAT is much higher than that of the other solvers.

Fig. 17 graphically shows the memory consumed by jSAT, [22] and zChaff solvers when solving instances generated from test case test13, which is the largest test case fully solved by all 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 is proportional to the bound. The same behavior has been observed in the other test cases, including those that jSAT failed to complete.

5.3.4. Performance analysis

The evaluation results shown in the previous section fulfill the expectations: jSAT is much more robust than the general-purpose QBF solvers and, in the same time, maintains their memory efficiency; on the other hand it is not as robust as the propositional SAT solvers.

The slower run-time of jSAT may be attributed to several factors. Firstly, the lack of explicit relations between all the states at the same time causes some implications, which a propositional SAT solver could make early in the decision process, to be made relatively late, after a number of state adjustment operations involving exploration of multiple transitions. Such a late discovery of implications allows an incorrect decision made on an earlier decision level to get through until an implication falsifying it is discovered, thus leading to a redundant exploration of the state-space in the meanwhile.

Secondly, the difference in the decision heuristics is known to affect the efficiency of the solution, and the decision heuristics of jSAT is much more restricted than those of the SAT solvers. The performance of the SAT solvers with decision heuristics similar to the one of the jSAT implementation was not evaluated as part of this work.

Another reason for the slower run-time of jSAT may be that many of the advanced optimizations present in the SAT solvers were not implemented in jSAT. Compared to [22], the evaluated jSAT implementation 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 state adjustment operations – those implemented by the procedures `AdvanceCurrentAndNextStates()` and `RetractCurrentAndNextStates()` in Fig. 12 and Fig. 14. The approach used for these operations, described earlier in Section 5.3.2, is simple but not very efficient. In fact, 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.

The performance analysis also showed that the conflict clauses built by the evaluated jSAT implementation systematically increase in length during the solution process. This follows from the way the implication graph is held in memory, as described in 5.3.2. Because of the information loss incurred by the state adjustment operation, more variables need to be included in the learned conflict clauses, since the knowledge that some of them were implied by others no longer exists.

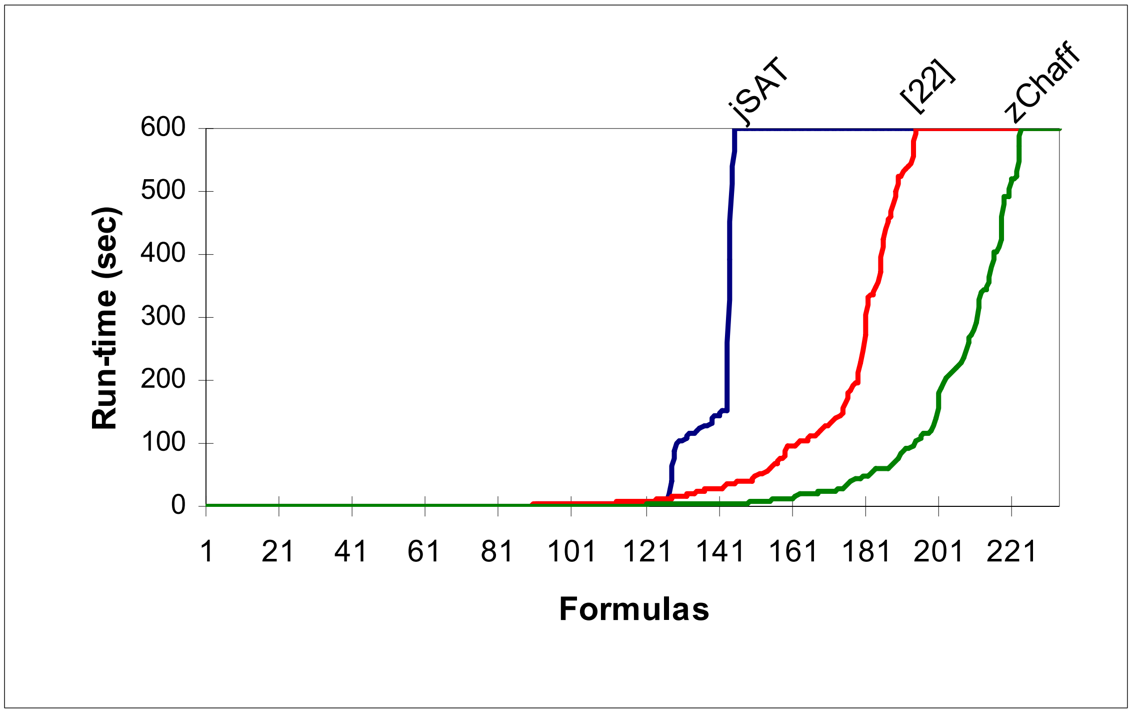


Fig. 16. Number of instances solved by each solver vs. the CPU time consumed.

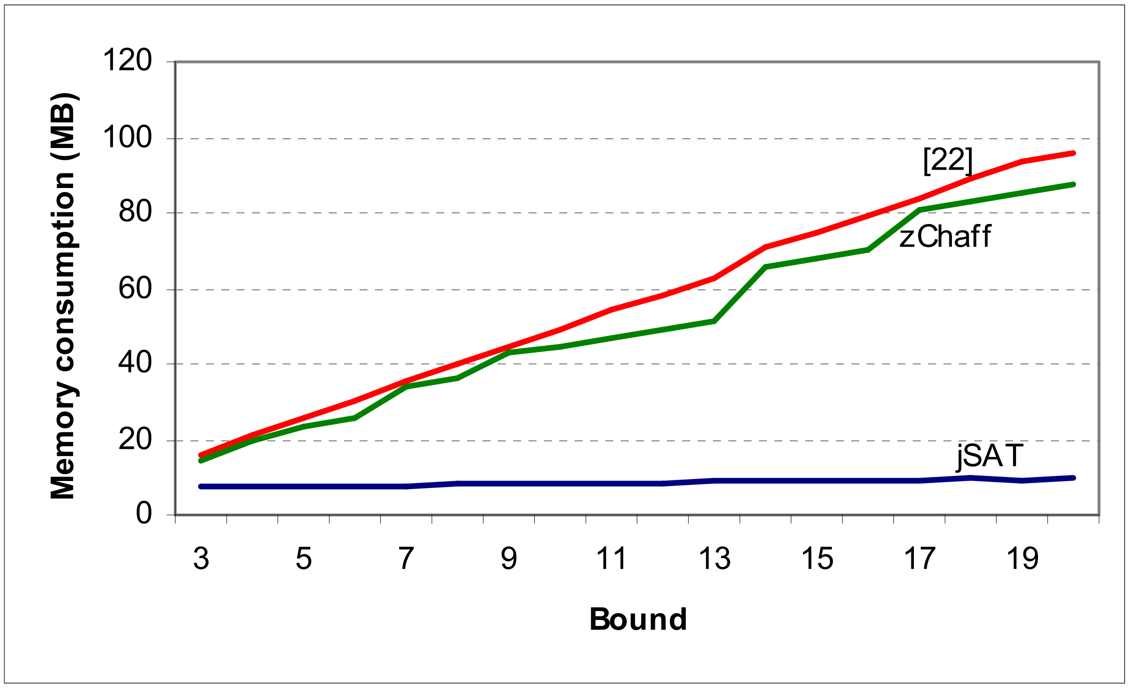


Fig. 17. Memory consumption of each solver on the instances generated for the test case test13.

Chapter 6

Conclusions and future research directions

As part of this work a study was performed in the following fields:

- the model checking domain, in general;
- existing model checking techniques, including BMC and unbounded SAT-based methods;
- DPLL-based SAT decision procedures; and
- DPLL-based QBF decision procedures.

The primary focus of this work was 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 benefit of the usage of QBF in BMC is the avoidance of the memory explosion problem occurring with other model checking methods, because it requires neither “unrolling” of the transition relation, nor the usage of propositional reasoning used in the existing model checking techniques. This work quantitatively shows that modern state-of-the-art general-purpose QBF solvers are still unable to handle in an efficient manner real-life instances of BMC problems encoded in QBF, but a special-purpose DPLL-based solver is feasible and capable of achieving much higher performance, significantly narrowing the performance gap between SAT-based BMC and QBF-based one.

The main contributions of this work are:

- a comparative study of the classical SAT-based BMC approach versus the approach using QBF formulations of BMC problems with the existing state-of-the-art general-purpose QBF solvers;
- publication of twenty QBF BMC encodings produced from real-life industrial BMC test cases of Intel® for the benefit of the academic research to develop and improve QBF solvers; and
- the development of a public-domain special-purpose DPLL-based QBF decision procedure, called jSAT, for the solution of QBF instances encoding BMC problems in form (16) and its experimental evaluation in comparison with the classical SAT-based BMC.

The following software was developed as part of this work:

- a simple bounded model checker, capable of reading a model from a file and producing formulas encoding BMC problems of different bounds in multiple approaches;
- jSAT algorithm implementation.

The results of this work have been partially published in [53][54].

A performance evaluation of jSAT shows that it achieves the expected memory savings, and succeeds to solve significantly more BMC 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.

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

- an improved data structure to enable more 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, e.g. restarting.

Besides improvements of the jSAT implementation, a number of additional research opportunities arise, which are described below.

Bounded model checking iteratively solves sub-problems for a range of bounds, increasing the bound from iteration to iteration. Clearly, within a system state graph any computation path of length *k* contains a path of length *k*-1. It is possible to reuse the information, found in the form of learned clauses, obtained during the solution of the instance for bound *k*-1 during the solution of the instance for bound *k*. This approach has been empirically shown as very useful in [56] in the context of SAT-based BMC. Since, unlike the classical SAT-based BMC, jSAT records learned clauses in terms of the state encoding variables only, it makes these clauses relevant and directly reusable for the solution of all the greater bounds as well, resulting with a significant search space reduction and speed-up of BMC.

Additionally, under an assumption that when BMC with bound *k* is performed all the bounds lower than *k* have been already checked, it is possible to avoid the exploration of those computation paths of length *k* which contain loops. Loops can be detected by jSAT because it has the knowledge of which are the states and their encoding variables. jSAT may retreat its DFS traversal when encountering a state that has been previously encountered, thus reducing the overall search space of the problem being solved.

The propositional BMC encoding used in the classical SAT-based method contains explicitly the relations between the states in the computation path of the system, as well as the explicit mentioning of what is the initial state and what is the final one. This is also partially the case with jSAT, where formula (17) contains an explicit specification of what the initial and the final states are. Avoiding the explicit specification of which is the final state may allow development of an unbounded depth-first search in the state graph of the system, enabling bounded model checking of a range of bounds in a single invocation of a jSAT-like algorithm. Explicit specification of what the final state is may be avoided by slightly changing the QBF formulation (16) of a single BMC problem:

$$\begin{aligned}&R_k(Z_0,Z_k) = \exists Z_1,\ldots,Z_{k-1} : I(Z_0) \wedge \\ &\left(\forall B : (Z_k \leftrightarrow B) \rightarrow F(B)\right) \wedge \\ &\forall U,V : \left(\bigvee_{i=0}^{k-1} (U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1}) \right) \rightarrow TR(U,V)\end{aligned} \tag{20}$$

Here the term F(Z_k) has been replaced with a formula specifying that a new variable *B* (standing for "bad state") is the final state. Analogously to the approach described in Section 5.1, it is possible to enhance jSAT to accept the following formula:

$$I(Z_0) \wedge TR(U,V) \wedge F(B) \tag{21}$$

The jSAT algorithm may then be enhanced to handle *B* as an alias of the last state in the path, similary to *U* and *V* that are aliases of a pair of neighboring states in the path, so that *B* may be associated with different states at different points of time. With such enhancements in place, the bound of the depth-first search in the state graph may be dynamically increased in run-time by introducing a new state and associating it with *B*, thus increasing the length of the paths being checked. Reuse of the learned information, as suggested above, is achieved trivially in this approach, because multiple BMC bounds are handled in the same algorithm invocation.

In fact, the approaches described above for the dynamic extension of the depth-first search and loop detection can be coupled together to enable a true symbolic depth-first search in the system state graph, enabling memory-efficient unbounded model checking.

The jSAT algorithm, as presented in this work, performs a forward DFS traversal of the system state space from the initial states to the final ones. Another interesting research direction is evaluation of a backward DFS traversal, e.g. by changing the decision strategy of jSAT to choose states in the opposite order. It may even be possible to further extend this direction by developing decision heuristics that make a non-linear order of state choices. Here it should be held in mind, however, that the current and the next state adjustment operations are relatively costly when performed frequently.

Chapter 7

Related work

jSAT, as presented in this work, implements a depth-first traversal of the system state graph using a general DPLL framework, in order to solve a QBF of a particular form in a memory-efficient manner, without replicating the tansition relation of the system multiple times. There is no other known solely DPLL-based algorithm implementing this approach.

The work most closely related to jSAT is in the domain of ATPG-based techniques in hardware verification. Automatic Test Pattern Generation (ATPG) is the task of generating a test for every fault in a hardware circuit according to some fault model. This field is closely related to hardware and has been driven primarily by specific applications in circuit testing. The relation of ATPG to formal hardware verification is briefly surveyed in [46].

One of the fault models considered in ATPG is the well-known *stuck-at fault model*, addressing the faults of a circuit when a specific wire gets stuck at a certain constant value due to manufacturing or other circuit defects. A test for stuck-at fault of a specific wire is obtained by finding assignments to the circuit inputs and state elements such that the fault is *controlled* at the chosen fault location, e.g. value 0 is produced for stuck-at-1 fault, and the fault is *observable*, i.e. a change on at least one output of the circuit can be observed as a result of the change of the value at the fault location. The controllability portion of this ATPG problem may be directly represented as a SAT problem;

accordingly, a SAT problem may be directly represented as an ATPG problem, if the CNF is interpreted as a two-level Boolean circuit.

Despite the similarity of concepts, due to historical reasons largely different terminology is used in the SAT and ATPG domains. Both techniques use backtrack search in the search space induced by the Boolean variables of the CNF or the circuit. The main difference between them, though, arises from the different problem formulation and representation: CNF or multi-level Boolean netlist. Techniques used for pruning the search space and making the implications vary as well, due to the difference in the problem representation. CNF-based SAT solvers enjoy the regularity of CNF to implement efficient implication techniques (e.g. BCP), conflict analysis, learning and non-chronological backtracking. Implication and conflict-based learning techniques for multi-level circuit ATPG are known to be less efficient. On the other hand, the CNF representation used in SAT lacks the structural circuit information that can be very useful. All in all, ATPG techniques pose an alternative to the usage of SAT in general and in model checking and hardware verification in particular.

The notion of Sequential ATPG (a.k.a. Sequential SAT) has been introduced as an alternative to the usage of "unrolled" transition relation of a system in SAT-based BMC. The combinational parts of a circuit model the transition relation of the system, since given the values at the inputs and the state elements of the circuit, those combinational parts determine the values for the outputs and the states of the circuit to assume in the next state. Sequential ATPG does not explicitly replicate the circuit to model a number of subsequent transitions, as opposed to the SAT-based BMC approach of "unrolling". Instead, the notion of time-frame is used to implicitly refer to the copy of the circuit at a certain time, and the algorithm works separately with different time-frames using the physically same representation of the circuit in memory, in a manner similar to jSAT. The Sequential ATPG backward reachability analysis algorithm starts with the setting of the desired final state after the last transition of the system, i.e. in the last time-frame. It then tries to find a justification for this setting by finding an assignment to the inputs and the state elements of the circuit in that time-frame that would cause the desired state to occur. A set of all such assignments constitutes an overall set of states from which the desired state is reachable. This set of states is the objective for the exploration of the previous time-frame. The algorithm proceeds until a legal initial state is covered in some time-frame, or a limit on the number of time-frames to explore is exhausted. This algorithm is a backward breadth-first (BFS) computation of the reachable states, somewhat similar to all-solutions SAT solvers, and potentially explodes in memory due to the complexity of the storage of the reachable states.

DFS exploration of the state space is also possible with the approach of Sequential SAT. To perform the DFS, the algorithm should not collect the justifying states of a time-frame all together, but try to justify them back in time as they are found. One such implementation is SATORI [13], which combines some ATPG and SAT techniques; particularly it operates on CNF in addition to the multi-level circuit for faster BCP. It has been reported outperforming state-of-the-art SAT solvers, mainly because of the availability of the circuit structure. Due to its DFS nature, it is also more memory efficient than the BFS approaches or SAT on "unrolled" model representation.

A few comparative studies of property checking using SAT versus ATPG have been published. For example, [47] shows that the two kinds of engines are comparable in performance. On the other hand, [48] shows a comparative study of BMC in hardware verification using SAT and ATPG techniques, claiming superiority of ATPG techniques. These conclusions are controversial, however, since some model checking practitioners (at least at Intel) claim the opposite observations in practice. The authors of [50] and [49] demonstrate an application of Sequential SAT based on SATORI for BMC, combining backward BFS and DFS traverses of the state space. It has been reported outperforming the classical SAT-based BMC, again mainly due to the availability of structural circuit information.

jSAT has not been evaluated against ATPG-based techniques. The memory efficiency trends observed with jSAT are similar to those of ATPG-based DFS approaches, but a number of differentiating qualities of jSAT vs. ATPG exist, namely:

- ATPG-based techniques are limited to backward search (BFS, DFS or a combination of those). jSAT, however, is more general and can be used to perform both backward and forward search. In fact, jSAT can work on different time-frames in any order, still being sound and complete.

- ATPG-based techniques are generally limited in learning capabilities so that the learned information does not involve states from distant time-frames. jSAT, on the other hand, is capable of producing conflict clauses involving any combinations of state encoding variables. In particular, since ATPG-based engines perform backward search and are limited in their learning, they are unable to learn from the constraints on the initial states, unlike jSAT. This is a considerable advantage for jSAT, since learning across time-frames can significantly prune the search space.

ATPG-based techniques are designed solely to cope with hardware verification. The main advantage of ATPG techniques that finds expression in their performance is the availability of the circuit structure to guide the search. jSAT, however, being based on the general DPLL framework is designed to solve any QBF instance of a particular form. It is possible that with circuit structure available, jSAT may achieve better results than shown in this work. Additionally, jSAT may benefit from its being based on DPLL, because the optimization techniques developed for DPLL-based SAT and/or QBF solvers can probably find their way to jSAT simpler than into ATPG-based techniques.

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. ”Memory Efficient All-Solutions SAT solver and its Application to Reachability”. 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 Application 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.

[28] L. Zhang, S. Malik. “The Quest for Efficient Boolean Satisfiability Solvers”. Computer Aided Verification (CAV), 2002

[29] M. Buro, H Kleine-Buning. “Report on a SAT competition”. Technical report, University of Paderborn, 1992

[30] J. Freeman. “Improvements to Propositional Satisfiability Search Algorithms”. PhD dissertation, Dept. of Computer and Information Science, Univ. of Pennsylvania, 1995

[31] E. Goldberg, Y. Novikov. “BerkMin: a Fast and Robust SAT Solver”. Design, Automation and Test in Europe Conference (DATE), 2002

[32] I. Lynce, J. Marques-Silva. "Efficient Data Structures for Backtrack Search SAT Solvers". International Conference on Theory and Applications of Satisfiability Testing (SAT), 2002

[33] L. Dryke, A. Frisch, I. Lynce, J.M. Silva, T. Walsh. "Comparing SAT Preprocessing Techniques". Proceedings of the 9th Workshop on Automated Reasoning, 2002

[34] E. Giunchiglia, M. Narizzano, A. Tacchella. "An Analysis of Backjumping and Trivial Truth in Quantified Boolean Formulas Satisfiability". Proceedings of the 7th Congress of the Italian Association for Artificial Intelligence on Advances in Artificial Intelligence, 2001

[35] E. Giunchiglia, M. Narizzano, A. Tacchella. "Learning for Quantified Boolean Logic Satisfiability". 18th National Conference on Artificial Intelligence, 2002

[36] L. Zhang, S. Malik. "Conflict Driven Learning in a Quantified Boolean Satisfiability Solver". International Conference on Computer Aided Design (ICCAD), 2002

[37] L. Zhang, S. Malik. "Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation". International Conference on Principles and Practice of Constraint Programming, 2002

[38] J. Rintanen. "Partial Implicit Unfolding in the Davis-Putnam Procedure for Quantified Boolean Formulae". Proceedings of the Artificial Intelligence on Logic for Programming, 2001

[39] J. Rintanen. "Improvements to the Evaluation of Quantified Boolean Formulae". Proceedings of the 16th International Joint Conference on Artificial Intelligence, 1999

[40] J. R. Burch, E. M. Clarke, K. McMillan, D. L. Dill, L. J. Hwang. "Symbolic Model Checking 10^20 States and Beyond". Proceedings of the 5th Annual Symposium on Logic in Computer Science, 1990.

[41] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, M.Y. Vardi. "Benefits of Bounded Model Checking at an Industrial Setting". Proceedings of the Computer Aided Verification (CAV), 2001

[42] D.A. Plaisted, A. Biere, Y. Zhu. "A Satisfiability Procedure for Quantified Boolean Formulae". Discrete Applied Mathematics Volume 130 (2003), pp. 291-328

[43] A.Bierre, C.Artho, V.Schuppan. "Liveness Checking as Safety Checking". Electronic Notes in Theoretical Computer Science, 2002

[44] E. Clarke, O. Grumberg, K. Hamaguchi. "Another Look at LTL Model Checking". Formal Methods in System Design, vol. 10, pp. 47-71, 1997

[45] Y. Feldman. "Parallel Multithreaded Satisfiability Solver: Design and Implementation". M.Sc. thesis, Tel-Aviv University, 2004.

[46] A. Biere, W. Kunz. "SAT and ATPG: Boolean Engines for Formal Hardware Verification". International Conference on Computer Aided Design (ICCAD), 2002.

[47] G. Parthasarathy, C.-Y. Huang, K.-T. Cheng. "An Analysis of ATPG and SAT Algorithms for Formal Verification". 6th IEEE International High-Level Design Validation and Test Workshop, 2001.

[48] D.G. Saab, J.A. Abraham, V.M. Vedula. "Formal Verification Using Bounded Model Checking: SAT versus Sequential ATPG Engines". 16th International Conference on VLSI Design, 2003.

[49] Parthasarathy, G., Iyer, M.K., Cheng, K.-T., Wang, L.-C. "Safety Property Verification Using Sequential SAT and Bounded Model Checking". Design & Test of Computers, Vol. 21 Issue 2, pp. 132-143, Mar-Apr 2004

[50] Parthasarathy, G., Iyer, M.K., Cheng, K.T., Wang, L.C. "Efficient Reachability Checking Using Sequential SAT". Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC), 2004

[51] J.P. Marques Silva. "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms". 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence, 1999

[52] zChaff SAT solver. URL: http://www.princeton.edu/~chaff/zchaff.html

[53] J. Katz, Z. Hanna, N. Dershowitz. "Space-Efficient Bounded Model Checking" (Extended Abstract). Proceedings of the Conference on Design, Automation and Test in Europe (DATE), 2005, Vol. 2, pp. 686 – 687

[54] N. Dershowitz, Z. Hanna, J. Katz. "Bounded Model Checking with QBF". International Conference on Theory and Applications of Satisfiability Testing (SAT), 2005.

[55] N. Dershowitz, Z. Hanna, A. Nadel. "A Clause-Based Heuristic for SAT Solvers". Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT), 2005.

[56] O. Strichman. "Accelerating Bounded Model Checking of Safety Properties". Formal Methods in System Design, Volume 24(1), pp. 5-24, 2004.

Annex A

jMC File Format

The simple bounded model checker developed as part of this work, called jMC, reads the model description from a text file in a format, the syntax of which is defined below. **Boldface** words are used to denote keywords, operators and punctuation required as part of the syntax. A vertical bar symbol | separates alternatives, square brackets [ ] denote optional items, and braces { } enclose items repeated one or more times.

The model description is given in the form of statements, which specify which are the state encoding variables in the model and the propositional formulas describing the initial states, the states violating the properties being checked (the bad states), and the transition relation. Additionally, an invariant formula may optionally be given to specify assumptions on the state encoding variables. The invariant formula is internally applied on the initial states and on the next state in the transition relation using conjunction with the two formulas.

Formulas are built from references to state encoding variables, simple binary and unary operators and the constants true and false. State encoding variables are used to specify the "current state of the system". A special "next-state" operator `'` following an identifier naming a state encoding variable is used to specify the value of the variable in the "next state of the system". To enable sharing of common sub-formulas, intermediate formulas can be given name using formula definition statements.

```
model_descrption ::= { statement }

statement ::=
    state_variable_declaration
    | transition_relation_declaration
    | initial_state_declaration
    | bad_state_declaration
    | invariant_declaration
    | formula_definition
```

```
state_variable_declaration ::=
    VAR identifier ;

transition_relation_declaration ::=
    TR expression ;

initial_state_declaration ::=
    INIT expression ;

bad_state_declaration ::=
    BAD expression ;

invariant_declaration ::=
    INVAR expression ;

formula_definition ::=
    DEFINE identifier = expression ;

expression ::=
      ( expression binary_operator expression )
    | ( expression )
    | ! expression
    | identifier
    | identifier'
    | true
    | false

identifier ::= [a-zA-Z0-9_$\[\]\.]{[a-zA-Z0-9_$\[\]\.]}

binary_operator ::= & | + | =

true ::= 1

false ::= 0
```

Annex B

jDIMACS File Format

The implementation of jSAT described in Chapter 5 reads formulas in jDIMACS format, which is a slightly modified version of DIMACS format. DIMACS format is commonly used to describe instances of SAT problems and contains the list of clauses that are part of the CNF formula to be solved. jDIMACS extends this simple format to include the knowledge of which are the state variables Z_0, …, Z_k and the variables U and V, since this knowledge is required for jSAT to operate.

The syntax of jDIMACS format is defined below, and utilizes the syntax of comments in the standard DIMACS format to provide additional information. **Boldface** words are used to denote keywords, operators and punctuation required as part of the syntax. A vertical bar symbol | separates alternatives, square brackets [ ] denote optional items, and braces { } enclose items repeated one or more times. The comment construct allowed by DIMACS format is excluded from the definition below for clarity, though supported by jDIMACS.

A jDIMACS file starts in the same way as a DIMACS file by specifying the total number of variables in the formula and the total number of clauses in the CNF body. Then, prior to the definition of the CNF body, a variable definition section should be present. The variable definition section specifies the number of states and the number of state encoding variables in those states. It then contains the ordered sequence of state encoding variables for each state, followed by the definition of the encoding variables of the current state U and the next state V.

```
jDimacs ::= header variable_definition body

header ::= p cnf number_of_variables number_of_clauses

variable_definition ::=
    c gp number_of_states number_of_state_variables
    state_variables { state_variables }
    U_variables V_variables

state_variables ::= c s { variable } 0

U_variables ::= c a { variable } 0

V_variables ::= c a { variable } 0

body ::= { clause } 0

clause ::= { literal }

number_of_variables ::= positive_number

number_of_clauses ::= positive_number

number_of_states ::= positive_number

number_of_state_variables ::= positive_number

variable ::= positive_number

literal ::= number

number ::= [-] positive_number

positive_number ::= [1-9] { [0-9] }
```

The following example shows an instance of a jSAT problem with three states and four encoding variables for each state. *Italicized* text are informative comments.

```
c This jSAT instance contains 266 variables and 382 clauses
p cnf 266 382

c There are 3 states in this jSAT instance, and each
c state is encoded by 4 variables
c gp 3 4

c The following lines define what variables encode each
c of the 3 states
c s 2 3 4 5 0
c s 259 260 261 262 0
c s 6 7 8 9 0

c The line below defines the variables that encode
c the current state U
```

```
c a 130 131 132 133 0
c The line below defines the variables that encode
c the next state V
c a 134 135 136 137 0

c The CNF body of the instance as a set of clauses
-16 6 0
-16 7 0
-6 -7 16 0
-17 8 0
-17 16 0
…………
```

Annex C

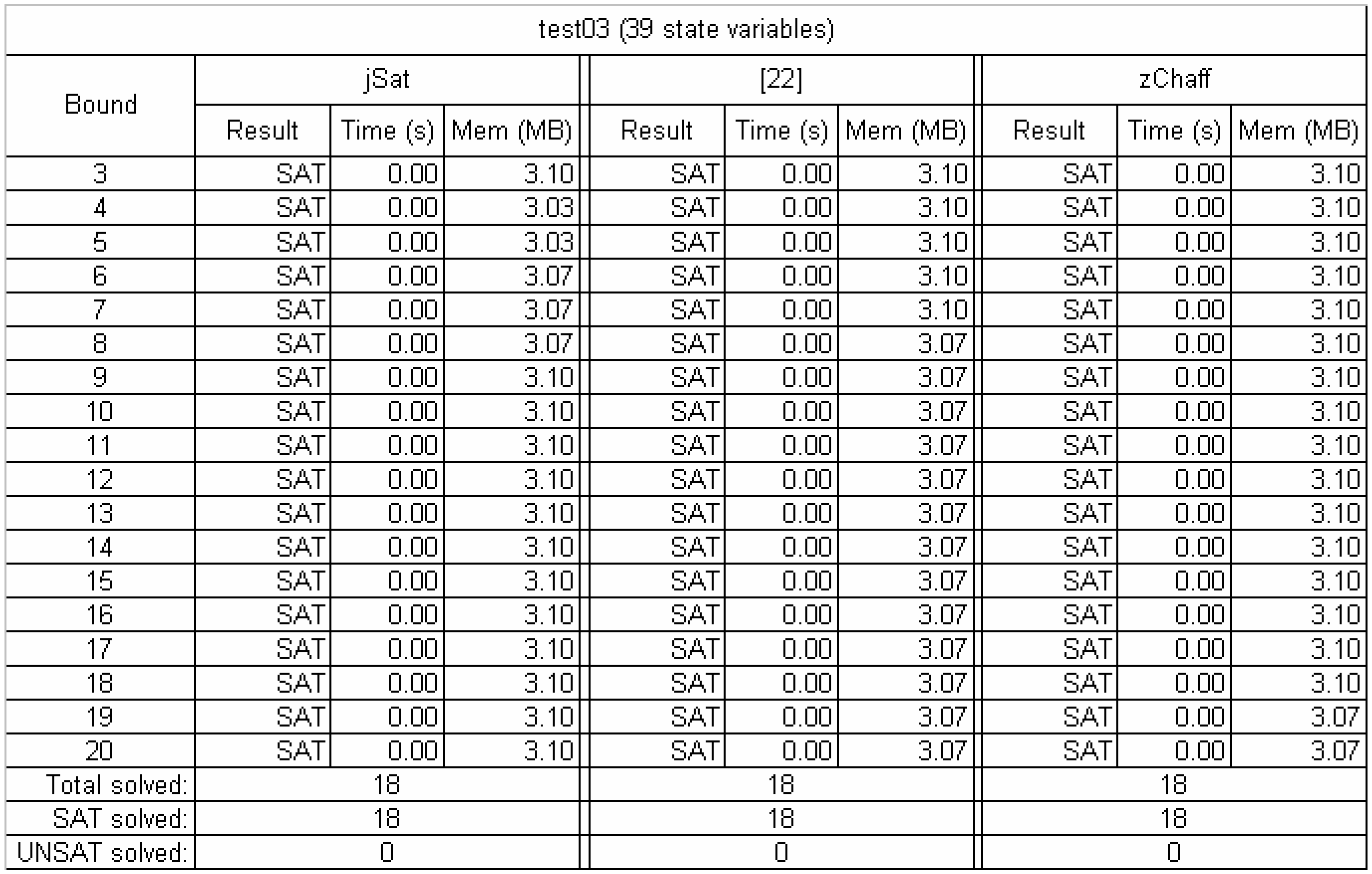
Detailed Benchmarking Results

The following tables show the detailed benchmarking results of the jSAT implementation described in Chapter 5, the SAT solver [22] on which the implementation of jSAT was based, and the SAT solver zChaff II. All the measurements were carried out on a dual Intel® Xeon™ 2.8 GHz Linux RedHat 7.1 workstation with 4GB of memory. Each solver was limited to solve every single instance within 10 minutes of run-time and within a 1GB memory envelope. The numbers were obtained by invoking every solver on every instance five times, dropping the least and the greatest measurements and taking the average of the remaining three. Time is reported in seconds, and memory consumption in MB.

test08 (10 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	UNSAT	0.00	3.10	UNSAT	0.00	3.10	UNSAT	0.00	3.10
4	UNSAT	0.00	3.10	UNSAT	0.00	3.10	UNSAT	0.00	3.07
5	UNSAT	0.00	3.07	UNSAT	0.00	3.10	UNSAT	0.00	3.07
6	UNSAT	0.00	3.07	UNSAT	0.00	3.10	UNSAT	0.00	3.07
7	UNSAT	0.00	3.07	UNSAT	0.00	3.10	UNSAT	0.00	3.07
8	UNSAT	0.17	2.60	UNSAT	0.00	3.10	UNSAT	0.00	3.07
9	UNSAT	1.00	2.40	UNSAT	0.00	3.10	UNSAT	0.00	3.07
10	UNSAT	3.70	2.40	UNSAT	0.00	3.10	UNSAT	0.00	3.07
11	UNSAT	3.78	2.40	UNSAT	0.00	3.10	UNSAT	0.00	3.07
12	UNSAT	4.02	2.40	UNSAT	0.00	3.10	UNSAT	0.00	3.07
13	UNSAT	4.25	2.50	UNSAT	0.00	3.10	UNSAT	0.00	3.07
14	UNSAT	4.52	2.50	UNSAT	0.00	3.10	UNSAT	0.00	3.07
15	UNSAT	4.74	2.50	UNSAT	0.00	3.10	UNSAT	0.00	3.07
16	UNSAT	5.24	2.50	UNSAT	0.00	3.10	UNSAT	0.00	3.07
17	UNSAT	9.50	2.50	UNSAT	0.00	3.10	UNSAT	0.00	3.07
18	UNSAT	87.93	2.50	UNSAT	0.00	3.10	UNSAT	0.00	3.10
19	(Time Out)	600.18	2.50	UNSAT	0.25	3.40	UNSAT	0.00	3.10
20	(Time Out)	600.07	2.50	UNSAT	0.25	3.40	UNSAT	0.00	3.10
Total solved:	16			18			18		
SAT solved:	0			0			0		
UNSAT solved:	16			18			18		

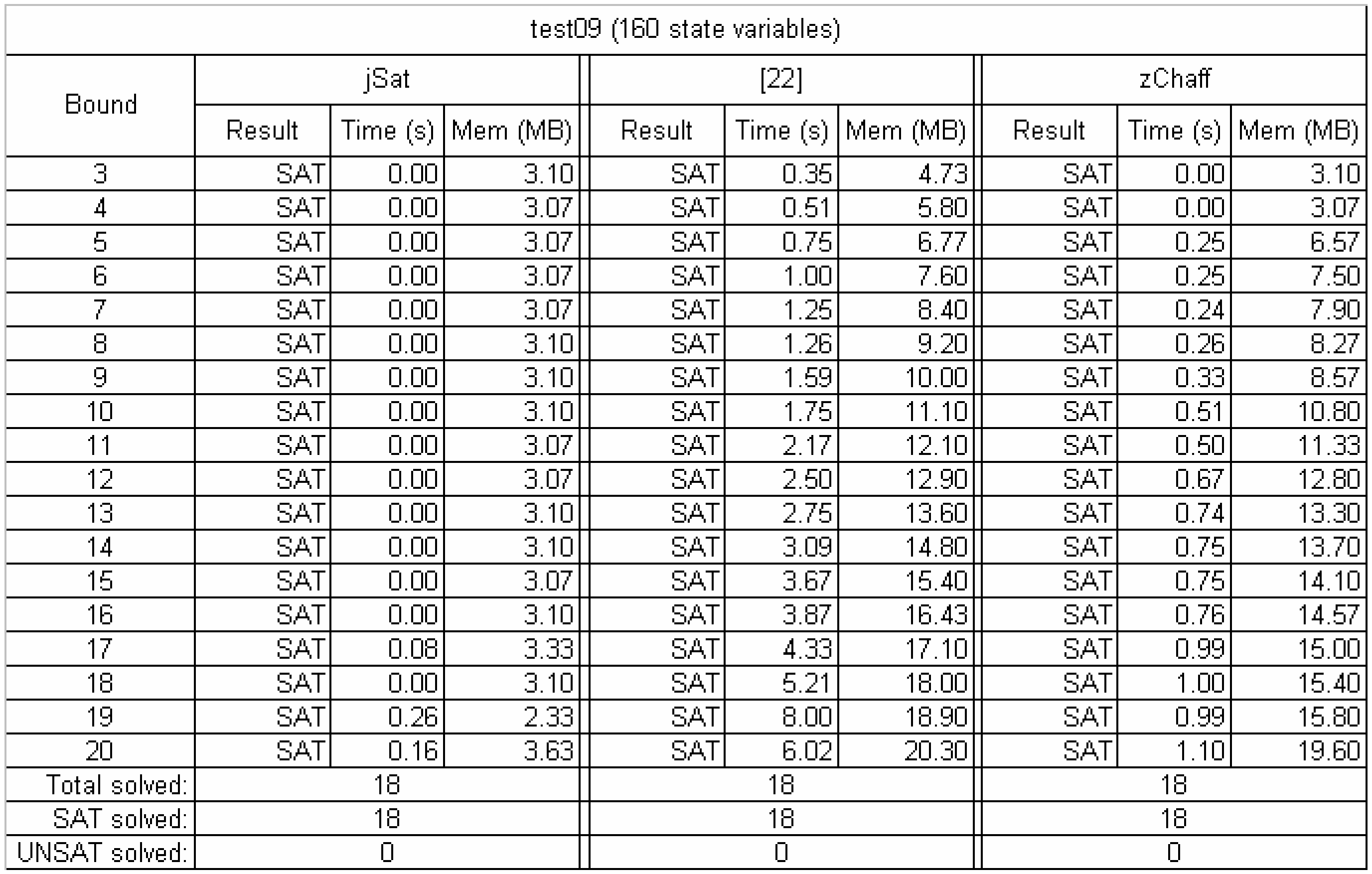
test12 (11 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	SAT	0.00	3.10	SAT	0.00	3.10	SAT	0.00	3.07
4	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
5	SAT	0.00	3.10	SAT	0.00	3.10	SAT	0.00	3.07
6	SAT	0.00	3.07	SAT	0.00	3.10	SAT	0.00	3.07
7	SAT	0.00	3.07	SAT	0.00	3.10	SAT	0.00	3.07
8	SAT	0.00	3.07	SAT	0.00	3.10	SAT	0.00	3.10
9	SAT	0.00	3.07	SAT	0.00	3.10	SAT	0.00	3.10
10	SAT	0.00	3.10	SAT	0.00	3.10	SAT	0.00	3.07
11	SAT	0.00	3.10	SAT	0.00	3.10	SAT	0.00	3.07
12	SAT	0.00	3.10	SAT	0.00	3.10	SAT	0.00	3.07
13	SAT	0.00	3.07	SAT	0.00	3.10	SAT	0.00	3.10
14	SAT	0.00	3.07	SAT	0.00	3.10	SAT	0.00	3.10
15	SAT	0.00	3.10	SAT	0.00	3.10	SAT	0.00	3.07
16	SAT	0.00	3.10	SAT	0.00	3.10	SAT	0.00	3.10
17	SAT	0.00	3.10	SAT	0.00	3.10	SAT	0.00	3.07
18	SAT	0.00	3.07	SAT	0.00	3.07	SAT	0.00	3.07
19	SAT	0.00	3.07	SAT	0.00	3.07	SAT	0.00	3.10
20	SAT	0.00	3.07	SAT	0.00	3.07	SAT	0.00	3.10
Total solved:	18			18			18		
SAT solved:	18			18			18		
UNSAT solved:	0			0			0		

test10 (12 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	UNSAT	0.00	3.07	UNSAT	0.00	3.10	UNSAT	0.00	3.07
4	UNSAT	0.00	3.07	UNSAT	0.00	3.10	UNSAT	0.00	3.07
5	UNSAT	0.00	3.10	UNSAT	0.00	3.10	UNSAT	0.00	3.07
6	UNSAT	0.00	3.10	UNSAT	0.00	3.07	UNSAT	0.00	3.07
7	UNSAT	0.00	3.10	UNSAT	0.00	3.07	UNSAT	0.00	3.10
8	UNSAT	0.00	3.07	UNSAT	0.00	3.07	UNSAT	0.00	3.10
9	UNSAT	0.00	3.07	UNSAT	0.00	3.07	UNSAT	0.00	3.10
10	UNSAT	0.00	3.07	UNSAT	0.26	3.30	UNSAT	0.00	3.07
11	UNSAT	0.00	3.10	UNSAT	0.26	3.37	UNSAT	0.00	3.07
12	UNSAT	0.00	3.10	UNSAT	1.27	5.60	UNSAT	0.00	3.10
13	UNSAT	0.00	3.10	UNSAT	0.50	3.93	UNSAT	0.00	3.10
14	UNSAT	0.00	3.07	UNSAT	1.52	5.53	UNSAT	0.00	3.10
15	UNSAT	0.25	2.60	UNSAT	0.26	3.60	UNSAT	0.07	3.20
16	UNSAT	0.25	2.60	UNSAT	1.51	5.90	UNSAT	0.26	3.67
17	UNSAT	0.25	2.60	UNSAT	0.75	4.30	UNSAT	0.25	3.70
18	UNSAT	0.25	2.60	UNSAT	0.50	4.10	UNSAT	0.25	3.70
19	UNSAT	0.25	2.60	UNSAT	0.76	4.50	UNSAT	0.50	4.10
20	UNSAT	0.25	2.60	UNSAT	1.00	4.77	UNSAT	0.50	4.10
Total solved:	18			18			18		
SAT solved:	0			0			0		
UNSAT solved:	18			18			18		



test03 (39 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	SAT	0.00	3.10	SAT	0.00	3.10	SAT	0.00	3.10
4	SAT	0.00	3.03	SAT	0.00	3.10	SAT	0.00	3.10
5	SAT	0.00	3.03	SAT	0.00	3.10	SAT	0.00	3.10
6	SAT	0.00	3.07	SAT	0.00	3.10	SAT	0.00	3.10
7	SAT	0.00	3.07	SAT	0.00	3.10	SAT	0.00	3.10
8	SAT	0.00	3.07	SAT	0.00	3.07	SAT	0.00	3.10
9	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
10	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
11	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
12	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
13	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
14	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
15	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
16	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
17	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
18	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.10
19	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.07
20	SAT	0.00	3.10	SAT	0.00	3.07	SAT	0.00	3.07
Total solved:	18			18			18		
SAT solved:	18			18			18		
UNSAT solved:	0			0			0		

test06 (160 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	UNSAT	41.15	3.60	UNSAT	0.41	5.00	UNSAT	0.25	5.20
4	(Time Out)	600.06	3.80	UNSAT	1.24	6.20	UNSAT	0.25	5.60
5	(Time Out)	600.09	4.00	UNSAT	2.50	7.40	UNSAT	0.91	7.10
6	(Time Out)	600.13	3.90	UNSAT	8.77	9.00	UNSAT	1.76	8.20
7	(Time Out)	600.16	3.90	UNSAT	13.79	10.70	UNSAT	4.77	9.00
8	(Time Out)	600.10	4.00	UNSAT	25.14	12.60	UNSAT	7.30	11.10
9	(Time Out)	600.10	4.00	UNSAT	61.32	19.90	UNSAT	12.29	13.30
10	(Time Out)	600.13	4.00	UNSAT	97.42	21.50	UNSAT	22.77	15.37
11	(Time Out)	600.06	4.10	UNSAT	178.80	26.97	UNSAT	37.50	20.10
12	(Time Out)	600.12	4.10	UNSAT	335.37	34.93	UNSAT	41.84	20.50
13	(Time Out)	600.05	4.13	UNSAT	330.39	40.07	UNSAT	58.32	22.30
14	(Time Out)	600.10	4.20	UNSAT	600.06	52.07	UNSAT	92.49	25.00
15	(Time Out)	600.15	4.33	(Time Out)	600.07	49.23	UNSAT	91.29	26.80
16	(Time Out)	600.14	4.40	(Time Out)	600.14	54.63	UNSAT	117.07	33.90
17	(Time Out)	600.07	4.40	(Time Out)	600.10	51.37	UNSAT	215.20	38.50
18	(Time Out)	600.12	4.40	(Time Out)	600.11	49.53	UNSAT	236.65	38.10
19	(Time Out)	600.17	4.50	(Time Out)	600.09	47.07	UNSAT	354.11	40.20
20	(Time Out)	600.08	4.40	(Time Out)	600.15	50.80	UNSAT	490.53	52.10
Total solved:	1			12			18		
SAT solved:	0			0			0		
UNSAT solved:	1			12			18		

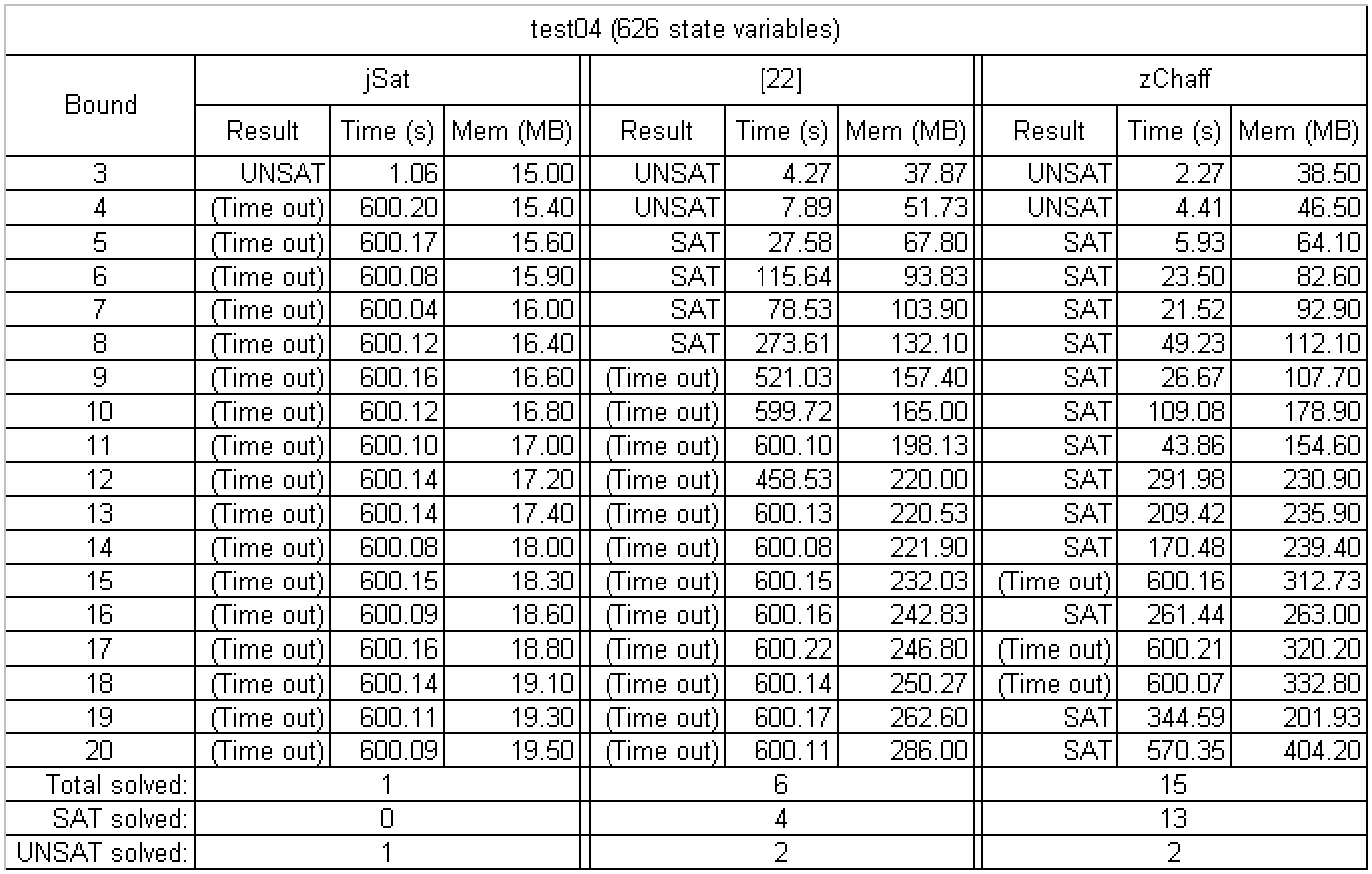


test09 (160 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	SAT	0.00	3.10	SAT	0.35	4.73	SAT	0.00	3.10
4	SAT	0.00	3.07	SAT	0.51	5.80	SAT	0.00	3.07
5	SAT	0.00	3.07	SAT	0.75	6.77	SAT	0.25	6.57
6	SAT	0.00	3.07	SAT	1.00	7.60	SAT	0.25	7.50
7	SAT	0.00	3.07	SAT	1.25	8.40	SAT	0.24	7.90
8	SAT	0.00	3.10	SAT	1.26	9.20	SAT	0.26	8.27
9	SAT	0.00	3.10	SAT	1.59	10.00	SAT	0.33	8.57
10	SAT	0.00	3.10	SAT	1.75	11.10	SAT	0.51	10.80
11	SAT	0.00	3.07	SAT	2.17	12.10	SAT	0.50	11.33
12	SAT	0.00	3.07	SAT	2.50	12.90	SAT	0.67	12.80
13	SAT	0.00	3.10	SAT	2.75	13.60	SAT	0.74	13.30
14	SAT	0.00	3.10	SAT	3.09	14.80	SAT	0.75	13.70
15	SAT	0.00	3.07	SAT	3.67	15.40	SAT	0.75	14.10
16	SAT	0.00	3.10	SAT	3.87	16.43	SAT	0.76	14.57
17	SAT	0.08	3.33	SAT	4.33	17.10	SAT	0.99	15.00
18	SAT	0.00	3.10	SAT	5.21	18.00	SAT	1.00	15.40
19	SAT	0.26	2.33	SAT	8.00	18.90	SAT	0.99	15.80
20	SAT	0.16	3.63	SAT	6.02	20.30	SAT	1.10	19.60
Total solved:	18			18			18		
SAT solved:	18			18			18		
UNSAT solved:	0			0			0		

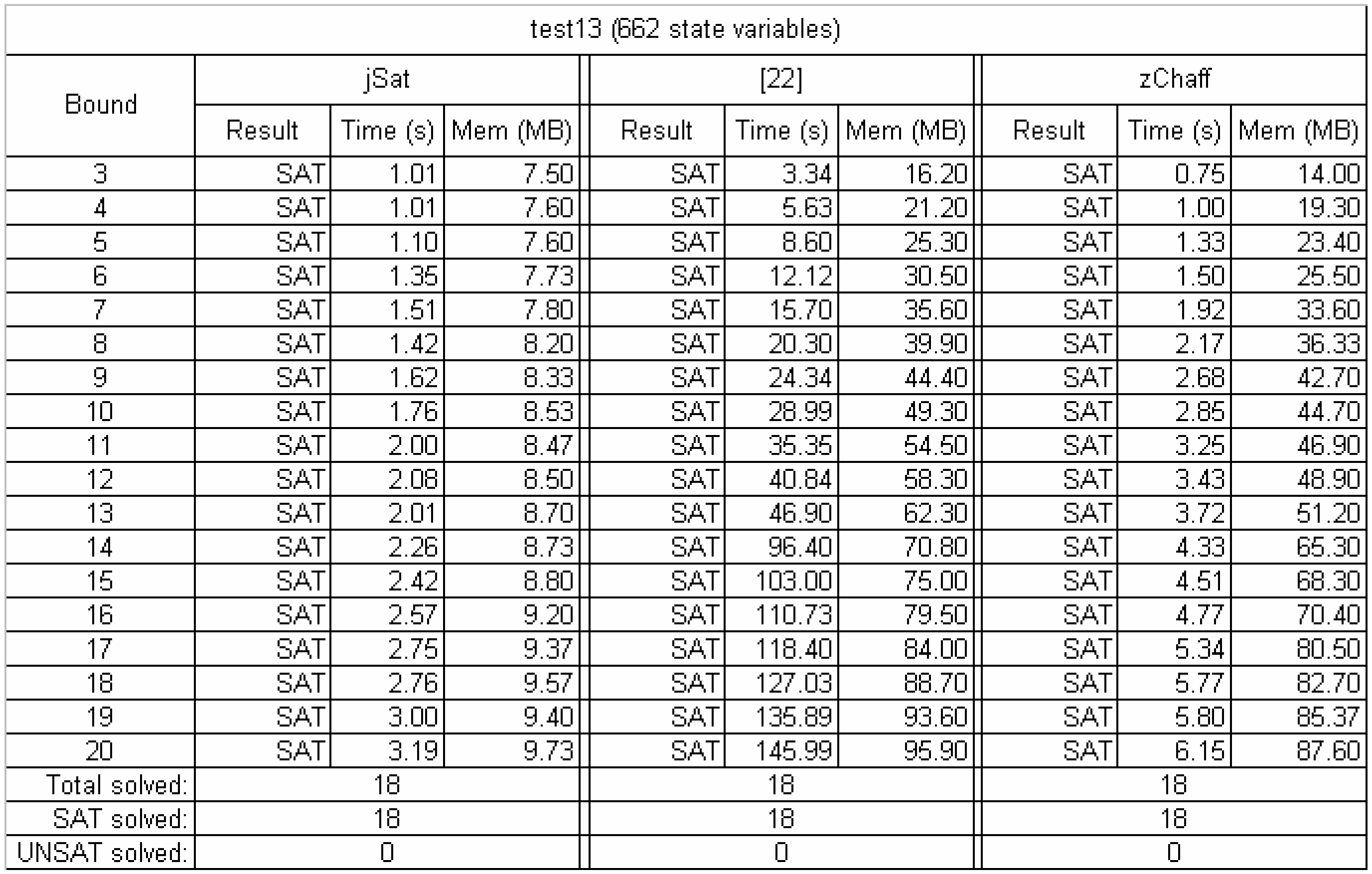
test05 (199 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	(Time Out)	600.14	15.30	UNSAT	0.49	5.73	UNSAT	0.25	5.40
4	(Time Out)	600.11	13.90	UNSAT	1.01	7.20	UNSAT	0.25	6.63
5	(Time Out)	600.10	14.03	UNSAT	1.50	8.43	UNSAT	0.49	8.00
6	(Time Out)	600.08	13.70	UNSAT	2.88	10.60	UNSAT	0.75	8.60
7	(Time Out)	600.16	13.80	UNSAT	5.48	13.73	UNSAT	1.42	10.97
8	(Time Out)	600.08	13.70	UNSAT	5.83	14.07	UNSAT	2.34	12.80
9	(Time Out)	600.10	13.83	UNSAT	10.72	17.73	UNSAT	2.92	13.40
10	(Time Out)	600.17	13.77	UNSAT	10.28	18.40	UNSAT	3.51	14.20
11	(Time Out)	600.09	13.70	UNSAT	14.44	22.50	UNSAT	4.86	15.50
12	(Time Out)	600.16	13.60	UNSAT	15.85	20.70	UNSAT	7.09	16.10
13	(Time Out)	600.19	13.23	UNSAT	22.69	23.53	UNSAT	6.44	16.40
14	(Time Out)	600.09	13.77	UNSAT	29.99	25.90	UNSAT	13.82	21.50
15	(Time Out)	600.10	13.70	UNSAT	39.49	29.90	UNSAT	12.41	23.50
16	(Time Out)	600.13	13.70	UNSAT	41.28	31.40	UNSAT	20.37	24.40
17	(Time Out)	600.10	13.53	UNSAT	52.02	33.27	UNSAT	19.94	25.27
18	(Time Out)	600.15	13.10	UNSAT	56.32	35.33	UNSAT	24.45	27.00
19	(Time Out)	600.14	14.03	UNSAT	67.26	37.23	UNSAT	21.94	27.37
20	(Time Out)	600.23	14.77	UNSAT	77.82	38.97	UNSAT	23.35	28.00
Total solved:	0			18			18		
SAT solved:	0			0			0		
UNSAT solved:	0			18			18		

test11 (220 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	UNSAT	0.25	3.60	UNSAT	0.25	5.10	UNSAT	0.00	3.07
4	UNSAT	5.76	3.80	UNSAT	0.25	5.60	UNSAT	0.00	3.07
5	UNSAT	6.00	3.90	UNSAT	0.50	7.50	UNSAT	0.26	7.70
6	UNSAT	390.97	4.00	UNSAT	0.76	8.70	UNSAT	0.51	8.37
7	SAT	102.10	4.10	SAT	1.77	10.40	SAT	0.50	9.00
8	SAT	105.61	4.20	SAT	2.26	11.60	SAT	0.50	11.10
9	SAT	109.32	4.20	SAT	3.01	12.60	SAT	1.26	12.80
10	SAT	116.29	4.17	SAT	3.51	13.90	SAT	1.26	13.60
11	SAT	117.72	4.37	SAT	3.52	14.90	SAT	0.75	13.80
12	SAT	120.34	4.40	SAT	5.00	16.20	SAT	0.51	14.10
13	SAT	123.63	4.40	SAT	5.77	17.20	SAT	0.50	15.00
14	SAT	127.12	4.47	SAT	6.27	18.90	SAT	0.51	15.40
15	SAT	128.85	4.50	SAT	5.26	19.70	SAT	0.50	16.00
16	SAT	132.36	4.57	SAT	7.93	21.57	SAT	0.75	19.90
17	SAT	153.93	4.67	SAT	8.45	22.90	SAT	4.76	23.50
18	SAT	151.95	4.63	SAT	8.89	23.70	SAT	0.74	21.20
19	SAT	143.96	4.60	SAT	53.63	29.70	SAT	4.51	26.70
20	SAT	144.53	4.80	SAT	8.26	25.60	SAT	0.76	24.07
Total solved:	18			18			18		
SAT solved:	14			14			14		
UNSAT solved:	4			4			4		

test04 (626 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	UNSAT	1.06	15.00	UNSAT	4.27	37.87	UNSAT	2.27	38.50
4	(Time out)	600.20	15.40	UNSAT	7.89	51.73	UNSAT	4.41	46.50
5	(Time out)	600.17	15.60	SAT	27.58	67.80	SAT	5.93	64.10
6	(Time out)	600.08	15.90	SAT	115.64	93.83	SAT	23.50	82.60
7	(Time out)	600.04	16.00	SAT	78.53	103.90	SAT	21.52	92.90
8	(Time out)	600.12	16.40	SAT	273.61	132.10	SAT	49.23	112.10
9	(Time out)	600.16	16.60	(Time out)	521.03	157.40	SAT	26.67	107.70
10	(Time out)	600.12	16.80	(Time out)	599.72	165.00	SAT	109.08	178.90
11	(Time out)	600.10	17.00	(Time out)	600.10	198.13	SAT	43.86	154.60
12	(Time out)	600.14	17.20	(Time out)	458.53	220.00	SAT	291.98	230.90
13	(Time out)	600.14	17.40	(Time out)	600.13	220.53	SAT	209.42	235.90
14	(Time out)	600.08	18.00	(Time out)	600.08	221.90	SAT	170.48	239.40
15	(Time out)	600.15	18.30	(Time out)	600.15	232.03	(Time out)	600.16	312.73
16	(Time out)	600.09	18.60	(Time out)	600.16	242.83	SAT	261.44	263.00
17	(Time out)	600.16	18.80	(Time out)	600.22	246.80	(Time out)	600.21	320.20
18	(Time out)	600.14	19.10	(Time out)	600.14	250.27	(Time out)	600.07	332.80
19	(Time out)	600.11	19.30	(Time out)	600.17	262.60	SAT	344.59	201.93
20	(Time out)	600.09	19.50	(Time out)	600.11	286.00	SAT	570.35	404.20
Total solved:	1			6			15		
SAT solved:	0			4			13		
UNSAT solved:	1			2			2		



test04 (626 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	UNSAT	1.06	15.00	UNSAT	4.27	37.87	UNSAT	2.27	38.50
4	(Time out)	600.20	15.40	UNSAT	7.89	51.73	UNSAT	4.41	46.50
5	(Time out)	600.17	15.60	SAT	27.58	67.80	SAT	5.93	64.10
6	(Time out)	600.08	15.90	SAT	115.64	93.83	SAT	23.50	82.60
7	(Time out)	600.04	16.00	SAT	78.53	103.90	SAT	21.52	92.90
8	(Time out)	600.12	16.40	SAT	273.61	132.10	SAT	49.23	112.10
9	(Time out)	600.16	16.60	(Time out)	521.03	157.40	SAT	26.67	107.70
10	(Time out)	600.12	16.80	(Time out)	599.72	165.00	SAT	109.08	178.90
11	(Time out)	600.10	17.00	(Time out)	600.10	198.13	SAT	43.86	154.60
12	(Time out)	600.14	17.20	(Time out)	458.53	220.00	SAT	291.98	230.90
13	(Time out)	600.14	17.40	(Time out)	600.13	220.53	SAT	209.42	235.90
14	(Time out)	600.08	18.00	(Time out)	600.08	221.90	SAT	170.48	239.40
15	(Time out)	600.15	18.30	(Time out)	600.15	232.03	(Time out)	600.16	312.73
16	(Time out)	600.09	18.60	(Time out)	600.16	242.83	SAT	261.44	263.00
17	(Time out)	600.16	18.80	(Time out)	600.22	246.80	(Time out)	600.21	320.20
18	(Time out)	600.14	19.10	(Time out)	600.14	250.27	(Time out)	600.07	332.80
19	(Time out)	600.11	19.30	(Time out)	600.17	262.60	SAT	344.59	201.93
20	(Time out)	600.09	19.50	(Time out)	600.11	286.00	SAT	570.35	404.20
Total solved:	1			6			15		
SAT solved:	0			4			13		
UNSAT solved:	1			2			2		

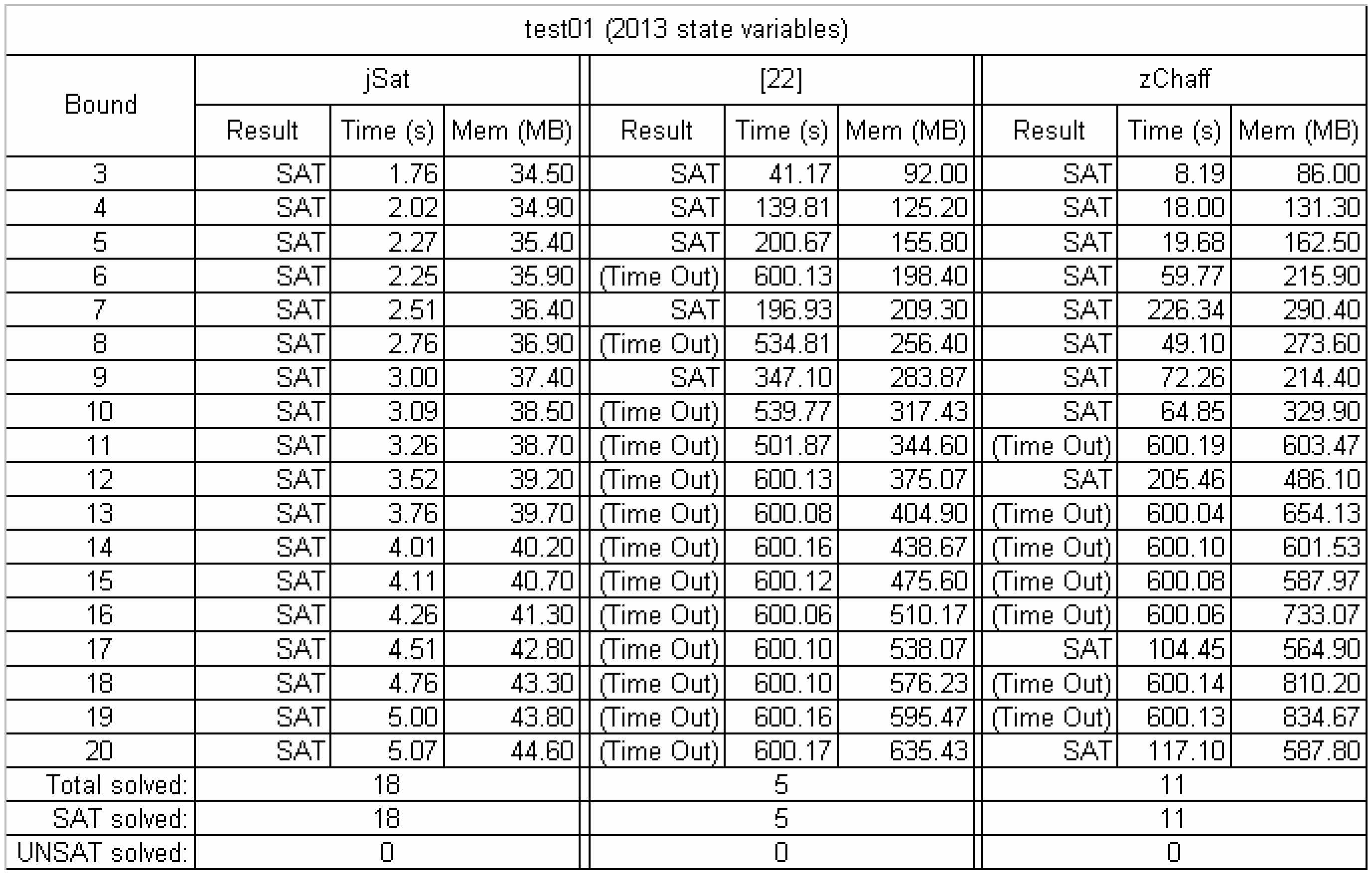


test13 (662 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	SAT	1.01	7.50	SAT	3.34	16.20	SAT	0.75	14.00
4	SAT	1.01	7.60	SAT	5.63	21.20	SAT	1.00	19.30
5	SAT	1.10	7.60	SAT	8.60	25.30	SAT	1.33	23.40
6	SAT	1.35	7.73	SAT	12.12	30.50	SAT	1.50	25.50
7	SAT	1.51	7.80	SAT	15.70	35.60	SAT	1.92	33.60
8	SAT	1.42	8.20	SAT	20.30	39.90	SAT	2.17	36.33
9	SAT	1.62	8.33	SAT	24.34	44.40	SAT	2.68	42.70
10	SAT	1.76	8.53	SAT	28.99	49.30	SAT	2.85	44.70
11	SAT	2.00	8.47	SAT	35.35	54.50	SAT	3.25	46.90
12	SAT	2.08	8.50	SAT	40.84	58.30	SAT	3.43	48.90
13	SAT	2.01	8.70	SAT	46.90	62.30	SAT	3.72	51.20
14	SAT	2.26	8.73	SAT	96.40	70.80	SAT	4.33	65.30
15	SAT	2.42	8.80	SAT	103.00	75.00	SAT	4.51	68.30
16	SAT	2.57	9.20	SAT	110.73	79.50	SAT	4.77	70.40
17	SAT	2.75	9.37	SAT	118.40	84.00	SAT	5.34	80.50
18	SAT	2.76	9.57	SAT	127.03	88.70	SAT	5.77	82.70
19	SAT	3.00	9.40	SAT	135.89	93.60	SAT	5.80	85.37
20	SAT	3.19	9.73	SAT	145.99	95.90	SAT	6.15	87.60
Total solved:	18			18			18		
SAT solved:	18			18			18		
UNSAT solved:	0			0			0		

test02 (914 state variables)

Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	(Time Out)	600.15	15.83	UNSAT	27.87	20.07	UNSAT	5.03	14.60
4	(Time Out)	600.18	16.00	UNSAT	37.68	26.37	UNSAT	10.25	22.90
5	(Time Out)	600.13	15.70	UNSAT	69.42	33.03	UNSAT	25.81	27.70
6	(Time Out)	600.11	16.03	UNSAT	104.85	37.40	UNSAT	28.00	29.20
7	(Time Out)	600.10	16.10	UNSAT	146.14	45.67	UNSAT	45.77	30.40
8	(Time Out)	600.12	15.77	UNSAT	182.51	51.57	UNSAT	61.22	43.30
9	(Time Out)	600.05	15.90	UNSAT	228.96	36.60	UNSAT	84.29	45.93
10	(Time Out)	600.12	16.03	UNSAT	370.93	64.87	UNSAT	96.10	48.80
11	(Time Out)	600.17	15.57	UNSAT	452.25	69.90	UNSAT	121.33	52.60
12	(Time Out)	600.06	15.90	UNSAT	422.60	74.47	UNSAT	139.27	55.20
13	(Time Out)	600.17	15.80	UNSAT	525.13	84.20	UNSAT	188.48	56.83
14	(Time Out)	600.15	16.33	UNSAT	554.27	89.83	UNSAT	220.92	58.90
15	(Time Out)	600.14	16.47	UNSAT	600.09	93.13	UNSAT	270.62	84.20
16	(Time Out)	600.12	16.37	(Time Out)	600.10	98.60	UNSAT	317.93	85.90
17	(Time Out)	600.08	16.60	(Time Out)	600.16	96.77	UNSAT	341.55	87.60
18	(Time Out)	600.21	17.07	(Time Out)	600.10	87.40	UNSAT	401.68	90.20
19	(Time Out)	600.06	16.87	(Time Out)	600.21	85.60	UNSAT	422.02	95.50
20	(Time Out)	600.18	17.53	(Time Out)	600.13	84.57	UNSAT	525.18	98.20
Total solved:	0			13			18		
SAT solved:	0			0			0		
UNSAT solved:	0			13			18		

test07 (1055 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	(Time Out)	600.15	18.23	SAT	4.95	21.30	SAT	1.43	21.60
4	(Time Out)	600.17	20.47	SAT	41.68	28.30	SAT	2.08	24.80
5	(Time Out)	600.10	26.43	SAT	12.15	33.90	SAT	4.27	29.80
6	(Time Out)	600.15	26.77	SAT	16.83	41.00	SAT	3.00	38.13
7	(Time Out)	600.14	28.33	SAT	21.88	46.50	SAT	3.76	45.70
8	(Time Out)	600.20	26.47	SAT	27.84	53.80	SAT	4.39	49.10
9	(Time Out)	600.13	24.57	SAT	35.66	60.30	SAT	10.03	52.70
10	(Time Out)	600.09	24.50	SAT	96.67	67.30	SAT	56.78	74.20
11	(Time Out)	600.11	24.07	(Time Out)	600.10	88.90	SAT	9.96	75.20
12	(Time Out)	600.17	24.27	(Time Out)	600.10	99.20	SAT	11.68	79.80
13	(Time Out)	600.10	24.27	SAT	112.22	90.67	SAT	58.79	110.80
14	(Time Out)	600.12	26.63	SAT	103.57	98.03	SAT	43.69	109.70
15	(Time Out)	600.09	26.67	SAT	128.28	101.70	SAT	75.08	119.50
16	(Time Out)	600.13	27.03	(Time Out)	600.08	127.20	SAT	19.35	70.40
17	(Time Out)	600.13	27.20	(Time Out)	600.21	137.30	SAT	493.52	149.90
18	(Time Out)	600.15	27.07	(Time Out)	600.07	145.30	SAT	404.61	153.90
19	(Time Out)	600.09	27.53	(Time Out)	600.15	153.43	SAT	520.88	165.20
20	(Time Out)	600.16	27.20	(Time Out)	600.21	150.40	(Time Out)	600.13	171.60
Total solved:	0			11			17		
SAT solved:	0			11			17		
UNSAT solved:	0			0			0		



test01 (2013 state variables)									
Bound	jSat			[22]			zChaff		
	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
3	SAT	1.76	34.50	SAT	41.17	92.00	SAT	8.19	86.00
4	SAT	2.02	34.90	SAT	139.81	125.20	SAT	18.00	131.30
5	SAT	2.27	35.40	SAT	200.67	155.80	SAT	19.68	162.50
6	SAT	2.25	35.90	(Time Out)	600.13	198.40	SAT	59.77	215.90
7	SAT	2.51	36.40	SAT	196.93	209.30	SAT	226.34	290.40
8	SAT	2.76	36.90	(Time Out)	534.81	256.40	SAT	49.10	273.60
9	SAT	3.00	37.40	SAT	347.10	283.87	SAT	72.26	214.40
10	SAT	3.09	38.50	(Time Out)	539.77	317.43	SAT	64.85	329.90
11	SAT	3.26	38.70	(Time Out)	501.87	344.60	(Time Out)	600.19	603.47
12	SAT	3.52	39.20	(Time Out)	600.13	375.07	SAT	205.46	486.10
13	SAT	3.76	39.70	(Time Out)	600.08	404.90	(Time Out)	600.04	654.13
14	SAT	4.01	40.20	(Time Out)	600.16	438.67	(Time Out)	600.10	601.53
15	SAT	4.11	40.70	(Time Out)	600.12	475.60	(Time Out)	600.08	587.97
16	SAT	4.26	41.30	(Time Out)	600.06	510.17	(Time Out)	600.06	733.07
17	SAT	4.51	42.80	(Time Out)	600.10	538.07	SAT	104.45	564.90
18	SAT	4.76	43.30	(Time Out)	600.10	576.23	(Time Out)	600.14	810.20
19	SAT	5.00	43.80	(Time Out)	600.16	595.47	(Time Out)	600.13	834.67
20	SAT	5.07	44.60	(Time Out)	600.17	635.43	SAT	117.10	587.80
Total solved:	18			5			11		
SAT solved:	18			5			11		
UNSAT solved:	0			0			0		