Bounded Model Checking

Armin Biere[1] Alessandro Cimatti[2] Edmund M. Clarke[3]
Ofer Strichman[3] Yunshan Zhu[4] ⋆

[1] Institute of Computer Systems, ETH Zurich, 8092 Zurich, Switzerland.
Email: biere@inf.ethz.ch
[2] Istituto per la Ricerca Scientifica e Technologica (IRST)
via Sommarive 18, 38055 Povo (TN), Italy. Email: cimatti@irst.itc.it
[3] Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA. Email: {emc,ofers}@cs.cmu.edu
[4] ATG, Synopsys, Inc., 700 East Middlefield Road, Mountain View, CA 94043, USA.
Email: yunshan@synopsys.com

Abstract. Symbolic model checking with Binary Decision Diagrams (BDDs) has been successfully used in the last decade for formally verifying finite state systems such as sequential circuits and protocols. Since its introduction in the beginning of the 90's, it has been integrated in the quality assurance process of several major hardware companies. The main bottleneck of this method is that BDDs may grow exponentially, and hence the amount of available memory restricts the size of circuits that can be verified efficiently. In this article we survey a technique called Bounded Model Checking (BMC), which uses a propositional SAT solver rather than BDD manipulation techniques. Since its introduction in 1999, BMC has been well received by the industry. It can find many logical errors in complex systems that can not be handled by competing techniques, and is therefore widely perceived as a complementary technique to BDD-based model checking. This observation is supported by several independent comparisons that have been published in the last few years.

1 Introduction

Techniques for automatic formal verification of finite state transition systems have developed in the last 12 years to the point where major chip design companies are beginning to integrate them in their normal quality assurance process. The most widely used of these methods is called *Model Checking*[11, 13]. In model checking, the design to be verified is modeled as a finite state machine, and the specification is formalized by writing *temporal logic* properties. The reachable states of the design are then traversed

⋆ This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Army Research Office (ARO) under grant DAAD19-01-1-0485, the Office of Naval Research (ONR), and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRC, ARO, NSF, ONR, NRL, the U.S. government or any other entity.

in order to verify the properties. In case the property fails, a counterexample is generated in the form of a sequence of states. In general, properties are classified to 'safety' and 'liveness' properties. While the former declares what should not happen (or equivalently, what should always happen), the latter declares what should eventually happen. A counterexample to safety properties is a trace of states, where the last state contradicts the property. A counterexample to liveness properties, in its simplest form, is a path to a loop that does not contain the desired state. Such a loop represents an infinite path that never reaches the specified state.

It is impossible to know whether the specification of a system is correct or complete – How can you know if what you wrote fully captures what you meant ? As a result, there is no such thing as a 'correct system'; it is only possible to check whether a system satisfies its specification or not. Moreover, even the most advanced model checkers are unable to verify all the desired properties of a system in a reasonable amount of time, due to the immense state-spaces of such systems. Model checking is often used for finding logical errors ('falsification') rather than for proving that they do not exist ('verification'). Users of model checking tools typically consider it as complimentary to the more traditional methods of testing and simulation, and not as an alternative. These tools are capable of finding errors that are not likely to be found by simulation. The reason for this is that unlike simulators, which examine a relatively small set of test cases, model checkers consider all possible behaviors or executions of the system. Also, the process of writing the temporal properties in a formal language can be very beneficial by itself, as it clarifies potential ambiguities in the specification.

The term Model Checking was coined by Clarke and Emerson [11] in the early eighties. The first model checking algorithms explicitly enumerated the reachable states of the system in order to check the correctness of a given specification. This restricted the capacity of model checkers to systems with a few million states. Since the number of states can grow exponentially in the number of variables, early implementations were only able to handle small designs and did not scale to examples with industrial complexity.

It was the introduction of *symbolic model checking* [9, 15] that made the first breakthrough towards wide usage of these techniques. In symbolic model checking, sets of states are represented implicitly using Boolean functions. For example, assume that the behavior of some system is determined by the two variables v_1 and v_2, and that $(11, 01, 10)$ are the three combinations of values that can be assigned to these variables in any execution of this system. Rather than keeping and manipulating this explicit list of states (as was done in explicit model checking), it is more efficient to handle a Boolean function that represents this set, e.g. $v_1 \vee v_2$. Manipulating Boolean formulas can be done efficiently with Reduced Ordered Binary Decision Diagrams [8] (ROBDD, or BDD for short), a compact, canonical graph representation of Boolean functions. The process works roughly as follows[1]: The set of initial states is represented as a BDD. The procedure then starts an iterative process, where at each step i, the set of states that can

[1] The exact details of this procedure depends on the property that is being verified. Here we describe the procedure for testing simple 'invariant' properties, which state that some proposition p has to hold invariantly in all reachable states. There is more than one way to perform this check.

first be reached in i steps from an initial state are added to the BDD. At each such step, the set of new states is intersected with the set of states that satisfy the negation of the property. If the resulting set is non-empty, it means that an error has been detected. This process terminates when the set of newly added states is empty or a an error is found. The first case indicates that the property holds, because no reachable state contradicts it. In the latter case, the model checker prints a counterexample. Note that termination is guaranteed, since there are only finitely many states.

The combination of symbolic model checking with BDDs [20, 15], pushed the barrier to systems with 10^{20} states and more [9]. Combining certain, mostly manual, abstraction techniques into this process pushed the bound even further. For the first time a significant number of realistic systems could be verified, which resulted in a gradual adoption of these procedures to the industry. Companies like Intel and IBM started developing their own in-house model checkers, first as experimental projects, and later as one more component in their overall quality verification process of their chip designs. Intel has invested significantly in this technology especially after the famous Pentium bug a few years ago.

The bottleneck of these methods is the amount of memory that is required for storing and manipulating BDDs. The Boolean functions required to represent the set of states can grow exponentially. Although numerous techniques such as decomposition, abstraction and various reductions have been proposed through the years to tackle this problem, full verification of many designs is still beyond the capacity of BDD based symbolic model checkers.

The technique that we describe in this article, called *Bounded Model Checking* (BMC), was first proposed by Biere et al. in 1999 [4]. It does not solve the complexity problem of model checking, since it still relies on an exponential procedure and hence is limited in its capacity. But experiments have shown that it can solve many cases that cannot be solved by BDD-based techniques. The converse is also true: there are problems that are better solved by BDD-based techniques. BMC also has the disadvantage of not being able to prove the absence of errors, in most realistic cases, as we will later explain. Therefore BMC joins the arsenal of automatic verification tools but does not replace any of them.

The basic idea in BMC is to search for a counterexample in executions whose length is bounded by some integer k. If no bug is found then one increases k until either a bug is found, the problem becomes intractable, or some pre-known upper bound is reached (this bound is called the *Completeness Threshold* of the design. We will elaborate on this point in section 5). The BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved by SAT methods rather than BDDs. SAT procedures do not suffer from the space explosion problem of BDD-based methods. Modern SAT solvers can handle propositional satisfiability problems with hundreds of thousands of variables or more.

Thus, although BMC aims at solving the same problem as traditional BDD-based symbolic model checking, it has two unique characteristics: first, the user has to provide a bound on the number of cycles that should be explored, which implies that the method is incomplete if the bound is not high enough. Second, it uses SAT techniques rather than BDDs. Experiments with this idea showed that if k is small enough (typi-

cally not more than 60 to 80 cycles, depending on the model itself and the SAT solver), it outperforms BDD-based techniques. Also, experiments have shown that there is little correlation between what problems are hard for SAT and what problems are hard for BDD based techniques. Therefore, the classes of problems that are known to be hard for BDDs, can many times be solved with SAT. If the SAT checkers are tuned to take advantage of the unique structure of the formulas resulting from BMC, this method improves even further [27]. A research published by Intel [14] showed that BMC has advantages in both capacity and productivity over BDD-based symbolic model checkers, when applied to typical designs taken from Pentium-4™. The improved productivity results from the fact that normally BDD based techniques need more manual guidance in order to optimize their performance. These and other published results with similar conclusions led most relevant companies, only three years after the introduction of BMC, to adopt it as a complementary technique to BDD-based symbolic model checking.

The rest of the article is structured as follows. In the next section we give a technical introduction to model checking and to the temporal logic that is used for expressing the properties. In section 3 we describe the bounded model checking problem. In the following section we describe the reduction of the BMC problem to Boolean satisfiability, including a detailed example. In section 5 we describe several methods for achieving completeness with BMC. In section 6 we describe some of the essential techniques underlying modern SAT solvers, and in section 7 we quote several experiments carried out by different groups, both from academia and industry, that compare these techniques to state of the art BDD-based techniques. We survey related work and detail our conclusions from the experiments in section 8.

2 Model checking

Model checking as a verification technique has three fundamental features. First, it is automatic; It does not rely on complicated interaction with the user for incremental property proving. If a property does not hold, the model checker generates a counterexample trace automatically. Second, the systems being checked are assumed to be finite[2]. Typical examples of finite systems, for which model checking has successfully been applied, are digital sequential circuits and communication protocols. Finally, temporal logic is used for specifying the system properties. Thus, model checking can be summarized as an algorithmic technique for checking temporal properties of finite systems.

As the reader may have deduced from the terminology we used in the introduction, we do not distinguish between the terms *design*, *system*, and *model*. An engineer working on real designs has to use a syntactic representation in a programming or hardware description language. Since we are only considering finite systems, the semantics of the engineer's design is usually some sort of a finite automaton. Independent of the concrete design language, this finite automaton can be represented by a *Kripke structure*,

[2] There is an ongoing interest in generalizing model checking algorithms to infinite systems, for example, by including real-time, or using abstraction techniques. In this article we will restrict the discussion to finite systems.

which is the standard representation of models in the model checking literature. It has its origin in modal logics, the generalization of temporal logics.

Formally, a Kripke structure M is a quadruple $M = (S, I, T, L)$ where S is the set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation and $L: S \to P(A)$ is the labeling function, where A is the set of atomic propositions, and $P(A)$ denotes the powerset over A. Labeling is a way to attach observations to the system: for a state $s \in S$ the set $L(s)$ is made of the atomic propositions that *hold* in s.

The notion of a Kripke structure is only a vehicle for illustrating the algorithms. It captures the semantics of the system under investigation. For a concrete design language, the process of extracting a Kripke structure from a given syntactic representation may not be that easy. In particular, the size of the system description and the size of the state space can be very different. For example, if we model a sequential circuit with a netlist of gates and flip-flops then the state space can be exponentially larger than the system description. A circuit implementing an n-bit counter illustrates this ratio: it can easily be implemented with $O(n)$ gates and $O(n)$ flip-flops, though the state space of this counter is 2^n. The exponential growth in the number of states poses the main challenge to model checking. This is also known as the *state explosion problem*.

The next step is to define the sequential behavior of a Kripke structure M. For this purpose we use *paths*. Each path π in M is a sequence $\pi = (s_0, s_1, \ldots)$ of states, given in an order that respects the transition relation of M. That is, $T(s_i, s_{i+1})$ for all $0 \leq i < |\pi| - 1$. If $I(s_0)$, i.e., s_0 is an initial state, then we say that the path is *initialized*. The length $|\pi|$ of π can either be finite or infinite. Note that in general some of the states may not be reachable, i.e., no initialized path leads to them. For $i < |\pi|$ we denote by $\pi(i)$ the i-th state s_i in the sequence and by $\pi_i = (s_i, s_{i+1}, \ldots)$ the suffix of π starting with state s_i. To simplify some technical arguments we assume that the set of initial states is non-empty. For the same reason we assume that the transition relation is *total*, i.e., each state has a successor state: for all $s \in S$ there exists $t \in S$ with $T(s, t)$.

As an example, consider the mutual exclusion problem of two processes competing for a shared resource. Pseudo code for this example can be found in Fig. 1. We assume that the processes are executed on a single computing unit in an interleaved manner. The **wait** statement puts a process into sleep. When all processes are asleep the scheduler tries to find a waiting condition which holds and reactivates the corresponding process. If all the waiting conditions are false the system stalls.

```
        process A                          process B
            forever                            forever
A.pc = 0        wait for B.pc = 0  B.pc = 0        wait for A.pc = 0
A.pc = 1        access shared resource  B.pc = 1   access shared resource
            end forever                        end forever
        end process                        end process
```

Fig. 1. Pseudo code for two processes A and B competing for a shared resource.

On an abstract level, each process has two program counter positions 0 and 1 with 1 representing the critical section. A process may only access the shared resource in the

critical section of its program. A state of the system is a pair of program counters and can be encoded as a binary vector $s \in S = \{0,1\}^2$ of length two. Thus $S = \{0,1\}^2$ is the set of states of the system. We assume that both processes start at program counter position 0, which implies that the set of initial states I consists of the single state represented by the Boolean vector 00. The transition relation consists of several possible transitions, according to the following two rules: the next state s' is the initial state 00 unless the current state is already the initial state; The initial state can transition forth and back to both 01 and 10. Thus, the transition relation $T \subseteq S^2 = \{0,1\}^4$ can be represented as the following set of bit strings:

$$\{0100, 1000, 1100, 0001, 0010\}$$

A graphical representation of this example in form of a Kripke structure is shown in Fig. 2. The initial state has an incoming edge without a source. The other edges correspond to one of the five transitions. Note that unreachable states, such as state 11 in this example, can only be removed after a *reachability analysis* has marked all reachable states. Accordingly the sequence $11, 00, 10, \ldots$ is a valid path of the Kripke structure,

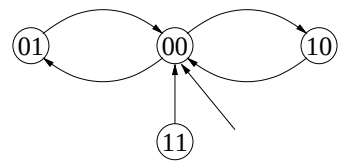

Fig. 2. A Kripke structure for two processes that preserve mutual exclusion.

but it is not initialized, since initialized paths start with the state 00. An example of an initialized path is the sequence $00, 01, 00, 10, 00, 01, \ldots$ where each process takes its turn to enter the critical region after the other process has left it.

Our example system is *safe* in the sense that the two processes obey the mutual exclusion property: at most one process can be in its critical region. A negative formulation of this property is that the state in which both processes are in their critical region is not reachable. Thus a simple model checking algorithm to check safety properties is to build the state transition graph and enumerate all reachable states through a graph search, starting from the set of initial states. Each visited state is analyzed in order to check whether it violates the safety property.

Now, assume that we add a faulty transition from 10 to 11. A depth first search, starting from the initial state 00 visiting 10 and then reaching 11 will show that the *bad* state 11 is reachable and thus the safety property fails. This path is a counterexample to the safety property that can help the user to debug the system.

What we have discussed so far is a typical *explicit* model checking algorithm for simple safety properties. It can be refined by building the Kripke structure on-the-fly: only after a state of the system is visited for the first time, the set of transitions is generated leaving this state. Once a bad state is found, the process terminates. This

technique is particularly useful if the number of reachable states is much smaller than $|S|$, the number of all states, which is often the case in practice.

Recall that safety properties describe invariants of a system, that is, that something bad does not happen. As we have seen, these properties can be checked by reachability analysis, i.e. by searching through the states graph and checking that each visited state does not violate the invariant. Also recall that in addition to safety properties, it is sometimes desirable to use liveness properties in order to check whether something good will eventually happen. In the mutual exclusion example, a natural question would be to ask whether each process will eventually enter its critical region. For the first process this means that the state 01 is eventually reached. More complicated liveness properties can specify repeatedly inevitable behavior, such as 'a request always has to be acknowledged'. To capture this nesting and mutual dependency of properties, temporal logic is used as a specification language.

Temporal logic is an extension of classical logic. In this article we concentrate on Propositional Linear Temporal Logic (PLTL, or LTL for short) as an extension of propositional logic. From propositional logic LTL inherits Boolean variables and Boolean operators such as negation $\neg$, conjunction $\wedge$, implication $\rightarrow$ etc. In addition to the Boolean connectives, LTL has temporal operators. First, there is the *next time* operator **X**. The formula $\mathbf{X}p$ specifies that property p holds at the next time step.

In Fig. 3(a) a path is shown for which $\mathbf{X}p$ holds. Each state is labeled with the atomic properties that hold in it. Fig. 3(b) depicts a path for which $\mathbf{X}p$ does not hold, because p holds in the first state but not in the next, second state. Now we can use this operator to

Fig. 3. Validity of next time operator in the formula $\mathbf{X}p$ along a path.

build larger temporal formulas. For instance $p \wedge \mathbf{X}\neg p$ holds iff p holds in the first state and p does not hold in the second. As usual $\neg$ is the Boolean negation operator. This formula is true for the path on Fig. 3(b) and fails for the path on Fig. 3(a). By nesting the operator X we can specify the behavior of the system up to a certain depth. For instance the formula $\mathbf{XX}p$ holds for both paths.

The next class of temporal operators that we discuss, allows specifying repeated unbounded behavior along an infinite path. The *Globally* operator **G** is used for safety properties. A formula $\mathbf{G}p$ *holds* along a path if p holds in all states of the path. Thus, it fails for the path in Fig. 3(b), since p does not hold in the second state. The safety property for our earlier example, the Kripke structure of Fig. 2, can be specified as $\mathbf{G}\neg(c_1 \wedge c_2)$, where c_i labels the states where process i is in its critical section. It literally can be translated into English as follows: for all states it is not the case that both c_1 and c_2 are true.

If *all* initialized paths of a Kripke structure satisfy a property, we say that the property holds for the Kripke structure. For instance by making the state 11 in Fig. 2 an

initial state, each path starting at 11 would be initialized and would violate $\mathbf{G}\neg(c_1 \wedge c_2)$ already in its first state. However since in our model 11 is not an initial state the property holds for the Kripke structure.

Finally we look at liveness properties. The simplest liveness operator is **F**, the *Finally* operator. The formula $\mathbf{F}p$ holds along a path if p holds somewhere on the path. Equivalently, it fails to hold if p stays unsatisfied along the whole path. For instance $\mathbf{F}p$ trivially holds in both paths of Fig. 3 since p is already satisfied in the first state. Similarly $\mathbf{F}\neg p$ holds for the path in Fig. 3(b), because p does not hold in the second state.

The liveness property for Fig. 2, which says that the first process will eventually reach its critical section, can be formalized as $\mathbf{F}c_1$. Since the system may loop between the initial state and the state 10 on the right, never reaching 01, this property does not hold. The initialized infinite path that starts with 00 and then alternates between 00 and 10 is a counterexample.

Now we can start to build more sophisticated specifications. The request / acknowledge property mentioned above is formulated as $\mathbf{G}(r \rightarrow \mathbf{F}a)$, where r and a are atomic propositions labeling states where a request and an acknowledge occurs, respectively. The same idea can be used to specify that a certain sequence of actions a_1, a_2, a_3 has to follow a guard g: $\mathbf{G}(g \rightarrow \mathbf{F}(a_1 \wedge \mathbf{F}(a_2 \wedge \mathbf{F}a_3)))$. Note that there may be an arbitrary, finite time interval (possibly empty) between the actions.

In this informal introduction to temporal logic, we will avoid a detailed explanation of the binary temporal operators *Until* (**U**) and *Release* (**R**). The reader is referred to [13] for more details. Also note that in the literature one can find an alternative notation for temporal operators, such as $\bigcirc p$ for $\mathbf{X}p$, $\Diamond p$ for $\mathbf{F}p$ and $\Box p$ for $\mathbf{G}p$.

The formal semantics of temporal formulas is defined with respect to paths of a Kripke structure. Let π be an infinite path of a Kripke structure M and let f be a temporal formula. We define recursively when f holds on π, written $\pi \models f$:

$$
\begin{array}{lll}
\pi \models p & \text{iff} & p \in L(\pi(0)) \\
\pi \models \neg f & \text{iff} & \pi \not\models f \\
\pi \models f \wedge g & \text{iff} & \pi \models f \text{ and } \pi \models g \\
\pi \models \mathbf{X}f & \text{iff} & \pi_1 \models f \\
\pi \models \mathbf{G}f & \text{iff} & \pi_i \models f \text{ for all } i \geq 0 \\
\pi \models \mathbf{F}f & \text{iff} & \pi_i \models f \text{ for some } i \geq 0 \\
\pi \models f\,\mathbf{U}g & \text{iff} & \pi_i \models g \text{ for some } i \geq 0 \text{ and } \pi_j \models f \text{ for all } 0 \leq j < i \\
\pi \models f\,\mathbf{R}g & \text{iff} & \pi_i \models g \text{ if for all } j < i,\ \pi_j \not\models f
\end{array}
$$

The semantics of the other Boolean operators such as disjunction and implication can be inferred from the above definition. As mentioned above we say that a temporal formula f holds for a Kripke structure M, written $M \models f$, iff $\pi \models f$ for all initialized paths π of M. Finally, we say that two temporal formulas f and g are *equivalent*, written $f \equiv g$ iff $M \models f \leftrightarrow M \models g$ for all Kripke structures M. With this notion, the semantics imply that $\neg\mathbf{F}\neg p \equiv \mathbf{G}p$. Thus, **F** and **G** are dual operators.

The standard technique for model checking LTL [19] is to compute the product of the Kripke structure with an automaton that represents the negation of the property (this automaton captures exactly the execution sequences that violate the LTL formula). Emptiness of the product automaton is an evidence of the correctness of the property. More details about this procedure can be found in [13].

3 Bounded model checking

The original motivation of bounded model checking was to leverage the success of SAT in solving Boolean formulas to model checking. During the last few years there has been a tremendous increase in reasoning power of SAT solvers. They can now handle instances with hundreds of thousands of variables and millions of clauses (we will elaborate more on how these solvers work in section 6). Symbolic model checkers with BDDs, on the other hand, can check systems with no more than a few hundred latches. Though clearly the number of latches and the number of variables cannot be compared directly, it seemed plausible that solving model checking with SAT could benefit the former.

A similar approach has been taken in tackling the planning problem in Artificial Intelligence [18]. Classical planning problems seek for a plan, i.e., a sequence of steps, to perform some task (e.g. position cubes one above the other in descending size under certain constraints on the intermediate states). As in BMC, the search for a plan is restricted to paths with some predetermined bound. The possible plans in a given bound are described by a SAT instance, which is polynomial in the original planning problem and the bound. Compared to model checking, deterministic planning is only concerned with simple safety properties: whether and how the goal state can be reached. In model checking we want to check liveness properties and nested temporal properties as well.

Since LTL formulas are defined over *all* paths, finding counterexamples corresponds to the question whether there *exists* a trace that contradicts them. If we find such a trace, we call it a *witness* for the property. For example, a counterexample to $M \models \mathbf{G}p$ corresponds to the question whether there exists a witness to $\mathbf{F}\neg p$. For clarity of presentation we will use *path quantifiers* **E** and **A** to denote whether the LTL formula is expected to be correct over all paths or only over some path. In other words, $M \models \mathbf{A}f$ means that M satisfies f over all initialized paths, and $M \models \mathbf{E}f$ means that there exists an initialized path in M that satisfies f. We will assume that the formula is given in *negation normal form* (NNF), in which negations are only allowed to occur in front of atomic propositions. Every LTL formula can be transformed to this form by using the duality of LTL operators and De-Morgan's laws.

The basic idea of bounded model checking, as was explained before, is to consider only a finite prefix of a path that may be a witness to an existential model checking problem. We restrict the length of the prefix by some bound k. In practice, we progressively increase the bound, looking for witnesses in longer and longer traces.

A crucial observation is that, though the prefix of a path is finite, it still might represent an infinite path if there is a *back loop* from the last state of the prefix to any of the previous states, as in Fig. 4(b). If there is no such back loop, as in Fig. 4(a), then the prefix does not say anything about the infinite behavior of the path beyond state s_k.

For instance, only a prefix with a back loop can represent a witness for $\mathbf{G}p$. Even if p holds along all the states from s_0 to s_k, but there is no back loop from s_k to a previous state, we cannot conclude that we have found a witness for $\mathbf{G}p$, since p might not hold at s_{k+1}.

(a) no loop (b) (k,l)-loop

Fig. 4. The two cases for a *bounded* path.

Definition 1. *For $l \leq k$ we call a path π a (k,l)-loop if $T(\pi(k),\pi(l))$ and $\pi = u \cdot v^\omega$ with $u = (\pi(0),\ldots,\pi(l-1))$ and $v = (\pi(l),\ldots,\pi(k))$[3]. We call π a k-loop if there exists $k \geq l \geq 0$ for which π is a (k,l)-loop.*

We will use the notion of k-loops in order to define the *bounded semantics* of model checking, i.e., semantics of model checking under bounded traces. The bounded semantics is an approximation to the unbounded semantics, which will allow us to define the bounded model checking problem. In the next section we will give a translation of a bounded model checking problem into a satisfiability problem.

In the bounded semantics we only consider a finite prefix of a path. In particular, we only use the first $k+1$ states $(s_0,\ldots,s_k)$ of a path to determine the validity of a formula along that path. If a path is a k-loop then we simply maintain the original LTL semantics, since all the information about this (infinite) path is contained in the prefix of length k.

Definition 2 (Bounded Semantics for a Loop). *Let $k \geq 0$ and π be a k-loop. Then an LTL formula f is valid along the path π with bound k (in symbols $\pi \models_k f$) iff $\pi \models f$.*

We now consider the case where π is not a k-loop. The formula $f := \mathbf{F}p$ is valid along π in the *unbounded* semantics if we can find an index $i \geq 0$ such that p is valid along the suffix π_i of π. In the bounded semantics the $(k+1)$-th state $\pi(k)$ does not have a successor. Therefore, unlike the unbounded case, we cannot define the bounded semantics recursively over *suffixes* (e.g. π_i) of π. We therefore introduce the notation $\pi \models_k^i f$, where i is the current position in the prefix of π, which means that the suffix π_i of π satisfies f, i.e., $\pi \models_k^i f$ implies $\pi_i \models f$.

Definition 3 (Bounded Semantics without a Loop). *Let $k \geq 0$, and let π be a path that is* not *a k-loop. Then an LTL formula f is* valid along π with bound k *(in symbols*

[3] The notation v^ω represents an infinite repetition of v.

$\pi \models_k f$) *iff* $\pi \models_k^0 f$ *where*

$$
\begin{array}{lll}
\pi \models_k^i p & \textit{iff} & p \in L(\pi(i)) \\
\pi \models_k^i \neg p & \textit{iff} & p \notin L(\pi(i)) \\
\pi \models_k^i f \wedge g & \textit{iff} & \pi \models_k^i f \textit{ and } \pi \models_k^i g \\
\pi \models_k^i f \vee g & \textit{iff} & \pi \models_k^i f \textit{ or } \pi \models_k^i g \\
\pi \models_k^i \mathbf{G} f & \textit{is always false} & \\
\pi \models_k^i \mathbf{F} f & \textit{iff} & \exists j,\, i \leq j \leq k.\ \pi \models_k^j f \\
\pi \models_k^i \mathbf{X} f & \textit{iff} & i < k \textit{ and } \pi \models_k^{i+1} f \\
\pi \models_k^i f \mathbf{U} g & \textit{iff} & \exists j,\, i \leq j \leq k.\ \pi \models_k^j g \textit{ and } \forall n, i \leq n < j.\ \pi \models_k^n f \\
\pi \models_k^i f \mathbf{R} g & \textit{iff} & \exists j,\, i \leq j \leq k.\ \pi \models_k^j f \textit{ and } \forall n, i \leq n < j.\ \pi \models_k^n g
\end{array}
$$

Note that if π is not a k-loop, then we say that $\mathbf{G} f$ is not valid along π in the bounded semantics with bound k since f might not hold along π_{k+1}. These constraints imply that for the bounded semantics the duality between $\mathbf{G}$ and $\mathbf{F}$ ($\neg \mathbf{F} f \equiv \mathbf{G} \neg f$) no longer holds.

Now we describe how the existential model checking problem ($M \models \mathbf{E} f$) can be reduced to a *bounded* existential model checking problem ($M \models_k \mathbf{E} f$). The basis for this reduction lies in the following two lemmas.

Lemma 1. *Let f be an LTL formula and π a path, then* $\pi \models_k f \Rightarrow \pi \models f$

Lemma 2. *Let f be an LTL formula and M a Kripke structure. If $M \models \mathbf{E} f$ then there exists $k \geq 0$ with $M \models_k \mathbf{E} f$*

Based on lemmas 1 and 2, we can now state the main theorem of this section. Informally, Theorem 1 says that if we take a sufficiently high bound, then the bounded and unbounded semantics are equivalent.

Theorem 1. *Let f be an LTL formula and M be a Kripke structure. Then $M \models \mathbf{E} f$ iff there exists $k \geq 0$ s.t. $M \models_k \mathbf{E} f$.*

4 Reducing bounded model checking to SAT

In the previous section we defined the semantics for bounded model checking. We now show how to reduce bounded model checking to propositional satisfiability. This reduction enables us to use efficient propositional SAT solvers to perform model checking.

Given a Kripke structure M, an LTL formula f and a bound k, we will construct a propositional formula $[\![M, f]\!]_k$. Let $s_0, \ldots, s_k$ be a finite sequence of states on a path π. Each s_i represents a state at time step i and consists of an assignment of truth values to the set of state variables. The formula $[\![M, f]\!]_k$ encodes constraints on $s_0, \ldots, s_k$ such that $[\![M, f]\!]_k$ is satisfiable iff π is a witness for f. The definition of formula $[\![M, f]\!]_k$ will be presented as three separate components. We first define a propositional formula $[\![M]\!]_k$ that constrains $s_0, \ldots, s_k$ to be a valid path starting from an initial state. We then

define the *loop condition*, which is a propositional formula that is evaluated to true only if the path π contains a loop. Finally, we define a propositional formula that constrains π to satisfy f.

Definition 4 (Unfolding of the Transition Relation). *For a Kripke structure M, $k \geq 0$*

$$[\![M]\!]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

The translation of an LTL formula depends on the shape of the path π. We define the propositional formula ${}_lL_k$ to be true if and only if there is a transition from state s_k to state s_l. By definition, ${}_lL_k$ is equal to $T(s_k, s_l)$. We use ${}_lL_k$ to define the loop condition L_k:

Definition 5 (Loop Condition). *The loop condition L_k is true if and only if there exists a back loop from state s_k to a previous state or to itself: $L_k := \bigvee_{l=0}^{k} {}_lL_k$*

Depending on whether a path is a k-loop (see Fig. 4), we have two different translations of a temporal formula f. First we consider the case where the path is a k-loop. We give a recursive translation of an LTL formula f for a k-loop path π. The translation of f recurses over its subterms and the states in π. The intermediate formula ${}_l[\![\cdot]\!]_k^i$ depends on three parameters: l, k and i. We use l for the start position of the loop, k for the bound, and i for the current position in π.

Definition 6 (Successor in a Loop). *Let k, l and i be non-negative integers s.t. $l, i \leq k$. Define the successor* $\mathrm{succ}(i)$ *of i in a (k,l)-loop as* $\mathrm{succ}(i) := i+1$ *for $i < k$ and* $\mathrm{succ}(i) := l$ *for $i = k$.*

Definition 7 (Translation of an LTL Formula for a Loop). *Let f be an LTL formula, $k, l, i \geq 0$, with $l, i \leq k$.*

$$
\begin{aligned}
{}_l[\![p]\!]_k^i &:= p(s_i) & {}_l[\![\mathbf{G} f]\!]_k^i &:= {}_l[\![f]\!]_k^i \wedge {}_l[\![\mathbf{G} f]\!]_k^{\mathrm{succ}(i)} \\
{}_l[\![\neg p]\!]_k^i &:= \neg p(s_i) & {}_l[\![\mathbf{F} f]\!]_k^i &:= {}_l[\![f]\!]_k^i \vee {}_l[\![\mathbf{F} f]\!]_k^{\mathrm{succ}(i)} \\
{}_l[\![f \vee g]\!]_k^i &:= {}_l[\![f]\!]_k^i \vee {}_l[\![g]\!]_k^i & {}_l[\![f \mathbf{U} g]\!]_k^i &:= {}_l[\![g]\!]_k^i \vee ({}_l[\![f]\!]_k^i \wedge {}_l[\![f \mathbf{U} g]\!]_k^{\mathrm{succ}(i)}) \\
{}_l[\![f \wedge g]\!]_k^i &:= {}_l[\![f]\!]_k^i \wedge {}_l[\![g]\!]_k^i & {}_l[\![f \mathbf{R} g]\!]_k^i &:= {}_l[\![g]\!]_k^i \wedge ({}_l[\![f]\!]_k^i \vee {}_l[\![f \mathbf{R} g]\!]_k^{\mathrm{succ}(i)}) \\
& & {}_l[\![\mathbf{X} f]\!]_k^i &:= {}_l[\![f]\!]_k^{\mathrm{succ}(i)}
\end{aligned}
$$

The translation in Definition 7 is linear with respect to the size of f and bound k if subterms are shared. A common technique for sharing subterms in propositional logic is to introduce new Boolean variables for subterms. Consider, for example, the formula $(a \wedge b) \vee (c \rightarrow (a \wedge b))$. We introduce a new variable x for the subterm $a \wedge b$, and transform the original formula into $(x \vee (c \rightarrow x)) \wedge (x \leftrightarrow (a \wedge b))$. The transformation clearly preserves satisfiability.

For the translation presented in Definition 7, a new propositional variable is introduced for each intermediate formula ${}_l[\![h]\!]_k^i$, where h is a subterm of the LTL formula

f and i ranges from 0 to k. The total number of new variables is $O(|f| \times k)$, where $|f|$ denotes the size of f. The size of the propositional formula ${}_l[\![f]\!]_k^0$ is also $O(|f| \times k)$.

For the case where π is not a k-loop, the translation can be treated as a special case of the k-loop translation. For Kripke structures with total transition relations, every finite path π can be extended to an infinite one. Since the property of the path beyond state s_k is unknown, we make a conservative approximation and assume all properties beyond s_k are false.

Definition 8 (Translation of an LTL Formula without a Loop).
Inductive Case: $\forall i \leq k$

$$
\begin{array}{ll}
[\![p]\!]_k^i := p(s_i) & [\![\mathbf{G} f]\!]_k^i := [\![f]\!]_k^i \wedge [\![\mathbf{G} f]\!]_k^{i+1} \\
[\![\neg p]\!]_k^i := \neg p(s_i) & [\![\mathbf{F} f]\!]_k^i := [\![f]\!]_k^i \vee [\![\mathbf{F} f]\!]_k^{i+1} \\
[\![f \vee g]\!]_k^i := [\![f]\!]_k^i \vee [\![g]\!]_k^i & [\![f \mathbf{U} g]\!]_k^i := [\![g]\!]_k^i \vee ([\![f]\!]_k^i \wedge [\![f \mathbf{U} g]\!]_k^{i+1}) \\
[\![f \wedge g]\!]_k^i := [\![f]\!]_k^i \wedge [\![g]\!]_k^i & [\![f \mathbf{R} g]\!]_k^i := [\![g]\!]_k^i \wedge ([\![f]\!]_k^i \vee [\![f \mathbf{R} g]\!]_k^{i+1}) \\
 & [\![\mathbf{X} f]\!]_k^i := [\![f]\!]_k^{i+1}
\end{array}
$$

Base Case:

$$[\![f]\!]_k^{k+1} := 0$$

Combining all components, the encoding of a bounded model checking problem in propositional logic is defined as follows.

Definition 9 (General Translation). *Let f be an LTL formula, M a Kripke structure and $k \geq 0$*

$$[\![M, f]\!]_k := [\![M]\!]_k \wedge \left(\left(\neg \mathrm{L}_k \wedge [\![f]\!]_k^0 \right) \vee \bigvee_{l=0}^{k} \left({}_l\mathrm{L}_k \wedge {}_l[\![f]\!]_k^0 \right) \right)$$

The left side of the disjunction is the case where there is no back loop and the translation without a loop is used. The right side represent all possible starting points l of a loop, and the translation for a (k,l)-loop is conjoined with the corresponding ${}_lL_k$ loop condition. The size of $[\![M, f]\!]_k$ is $O(|f| \times k \times |M|)$, where $|M|$ represents the size of the syntactic description of the initial state I and the transition relation T.

The translation scheme guarantees the following theorem, which we state without proof:

Theorem 2. *$[\![M, f]\!]_k$ is satisfiable iff $M \models_k \mathbf{E} f$.*

Thus, the reduction of bounded model checking to SAT is sound and complete with respect to the bounded semantics.

Example 1. Let us consider the mutual exclusion example in Fig. 2. Each state s of the system M is represented by two bit variables. We use $s[1]$ for the high bit and $s[0]$ for the low bit.

The initial state is represented as follows,

$$I(s) := \neg s[1] \wedge \neg s[0]$$

The transition relation is represented as follows,

$$T(s,s') := (\neg s[1] \wedge (s[0] \leftrightarrow \neg s'[0])) \vee (\neg s[0] \wedge (s[1] \leftrightarrow \neg s'[1])) \vee (s[0] \wedge s[1] \wedge \neg s'[1] \wedge \neg s'[0])$$

We now add a faulty transition from state 10 to state 11. We denote by T_f the new faulty transition relation.

$$T_f(s,s') := T(s,s') \vee (s[1] \wedge \neg s[0] \wedge s'[1] \wedge s'[0])$$

Consider the safety property that at most one process can be in the critical region at any time. The property can be represented as $\mathbf{G}\neg p$, where p is $s[1] \wedge s[0]$. Using BMC, we attempt to find a counterexample of the property, or, in other words, look for a witness for $\mathbf{F}p$. The existence of such a witness indicates that the mutual exclusion property is violated by M. If, on the other hand, no such witness can be found, it means that this property holds up to the given bound.

Let us consider a case where the bound $k = 2$. Unrolling the transition relation results in the following formula:

$$[\![M]\!]_2 := I(s_0) \wedge T_f(s_0,s_1) \wedge T_f(s_1,s_2)$$

The loop condition is:

$$L_2 := \bigvee_{l=0}^{2} T_f(s_2,s_l)$$

The translation for paths without loops is:

$$\begin{array}{lcllcl} [\![\mathbf{F}p]\!]_2^0 & := & p(s_0) \vee [\![\mathbf{F}p]\!]_2^1 & [\![\mathbf{F}p]\!]_2^1 & := & p(s_1) \vee [\![\mathbf{F}p]\!]_2^2 \\ [\![\mathbf{F}p]\!]_2^2 & := & p(s_2) \vee [\![\mathbf{F}p]\!]_2^3 & [\![\mathbf{F}p]\!]_2^3 & := & 0 \end{array}$$

We can introduce a new variable for each intermediate formula $[\![\mathbf{F}p]\!]_2^i$. Alternatively, we can substitute all intermediate terms and obtain the following formula.

$$[\![\mathbf{F}p]\!]_2^0 := p(s_0) \vee p(s_1) \vee p(s_2)$$

The translation with loops can be done similarly. Putting everything together we get the following Boolean formula:

$$[\![M, \mathbf{F}p]\!]_2 := [\![M]\!]_2 \wedge \left(\left(\neg \mathrm{L}_2 \wedge [\![\mathbf{F}p]\!]_2^0 \right) \vee \bigvee_{l=0}^{2} \left({}_l\mathrm{L}_2 \wedge {}_l[\![\mathbf{F}p]\!]_2^0 \right) \right) \qquad (1)$$

Since a finite path to a bad state is sufficient for falsifying a safety property, the loop condition in the above formula may be omitted. This will result in the following formula:

$$\begin{array}{lll} [\![M, \mathbf{F}p]\!]_2 := & [\![M]\!]_2 \wedge [\![\mathbf{F}p]\!]_2^0 = & \\ & I(s_0) \wedge T_f(s_0,s_1) \wedge T_f(s_1,s_2) \wedge (p(s_0) \vee p(s_1) \vee p(s_2)) & \end{array}$$

The assignment 00, 10, 11 satisfies $[\![M, \mathbf{F}p]\!]_2$. This assignment corresponds to a path from the initial state to the state 11 that violates the mutual exclusion property. □

5 Techniques for Completeness

Given a model checking problem $M \models \mathbf{E}f$, a typical application of BMC starts at bound 0 and increments the bound until a witness is found. This represents a partial decision procedure for model checking problems. If $M \models \mathbf{E}f$, a witness of finite length k exists, and the procedure terminates at length k. If $M \not\models \mathbf{E}f$, however, the outlined procedure does not terminate. Although the strength of BMC is in detection of errors, it is desirable to build a complete decision procedure based on BMC for obvious reasons. For example, BMC may be used to clear a module level proof obligation which may be as assumption for another module. A missed counterexample in a single module may have the unpleasant consequence of breaking the entire proof. In such compositional reasoning environments, completeness becomes particularly important.

In this section, we will highlight three techniques for achieving completeness with BMC. For unnested properties such as $\mathbf{G}p$ and $\mathbf{F}p$, we determine in section 5.1 the maximum bound k that the BMC formula should be checked with in order to guarantee that the property holds. This upper bound is called the Completeness Threshold. For liveness properties, we show an alternative path to completeness in section 5.2. The alternative method is based on a semi-decision procedure for $\mathbf{AF}p$ combined with a semi decision procedure for $\mathbf{EG}p$. Finally, in section 5.3, we show how for safety properties completeness can be achieved with induction based on strengthening inductive invariants.

5.1 The completeness threshold

For every finite state system M, a property p, and a given translation scheme, there exists a number $\mathcal{CT}$, such that the absence of errors up to cycle $\mathcal{CT}$ proves that $M \models p$. We call $\mathcal{CT}$ the *Completeness Threshold* of M with respect to p and the translation scheme.

The completeness threshold for $\mathbf{G}p$ formulas is simply the minimal number of steps required to reach all states. We call this the *reachability diameter* and formally define it as follows:

Definition 10 (Reachability Diameter). *The reachability diameter $rd(M)$ is the minimal number of steps required for reaching all reachable states:*

$$\begin{array}{l} rd(M) := \min\{i | \forall s_0,\ldots,s_n.\ \exists s'_0,\ldots,s'_t, t \leq i. \\ I(s_0) \wedge \bigwedge_{j=0}^{n-1} T(s_j,s_{j+1}) \rightarrow (I(s'_0) \wedge \bigwedge_{j=0}^{t-1} T(s'_j,s'_{j+1}) \wedge s'_t = s_n)\} \end{array} \quad (2)$$

Formula (2) simply states that every state that is reachable in n steps (left side of the implication) can also be reached in i steps (right side of the implication). In other words, $rd(M)$ is the longest 'shortest path' from an initial state to any reachable state. This definition leaves open the question of how large should n be. One option is to simply take the worst case, i.e. $n = 2^{|V|}$, where V is the set of variables defining the states of M. A better option is to take $n = i+1$ and check whether every state that can be reached in $i+1$ steps, can be reached sooner:

$$\begin{array}{l} rd(M) := \min\{i | \forall s_0,\ldots,s_{i+1}.\ \exists s'_0,\ldots,s'_i. \\ I(s_0) \wedge \bigwedge_{j=0}^{i} T(s_j,s_{j+1}) \rightarrow (I(s'_0) \wedge \bigwedge_{j=0}^{i-1} T(s'_j,s'_{j+1}) \wedge \bigvee_{j=0}^{i} s'_j = s_{i+1})\} \end{array} \quad (3)$$

In Formula (3), the sub formula to the left of the implication represent an $i+1$ long path, and the sub-formula to the right of the implication represents an i long path. The disjunction in the end of the right hand side forces the $i+1$ state in the longer path to be equal to one of the states in the shorter path.

Both equations 2 and 3 include an alternation of quantifiers, and are hence hard to solve for realistic models. As an alternative, it is possible to compute an over approximation of $rd(M)$ with a SAT instance. This approximation was first defined in [4] as the *recurrence diameter*, and we now adapt it to the reachability diameter:

Definition 11 (Recurrence Diameter for Reachability). *The Recurrence Diameter for Reachability with respect to a model M, denoted by $rdr(M)$, is the longest loop-free path in M starting from an initial state:*

$$rdr(M) := max\{i|\ \exists s_0 \ldots s_i.\ I(s_0) \wedge \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1}) \wedge \bigwedge_{j=0}^{i-1} \bigwedge_{k=j+1}^{i} s_j \neq s_k\} \tag{4}$$

$rdr(M)$ is clearly an over-approximation of $rd(M)$, because every shortest path is a loop-free path.

The question of how to compute $\mathcal{CT}$ for other temporal properties is still open. Most safety properties used in practice can be reduced to some $\mathbf{G}p$ formula, by computing p over a product of M and some automaton, which is derived from the original property. Therefore computing $\mathcal{CT}$ for these properties is reduced to the problem of computing $\mathcal{CT}$ of the new model with respect to a $\mathbf{G}p$ property.

5.2 Liveness

In the discussion of bounded model checking so far, we have focused on existentially quantified temporal logic formulas. To verify an existential LTL formula against a Kripke structure, one needs to find a witness. As explained before, this is possible because if a witness exists, it can be characterized by a finite sequence of states. In the case of liveness, the dual is also true: if a proof of liveness exists, the proof can be established by examining all finite sequences of length k starting from initial states (note that for a proof we need to consider all paths rather than search for a single witness).

Definition 12 (Translation for Liveness Properties).

$$[\![\, M, \mathbf{AF}p \,]\!]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \rightarrow \bigvee_{i=0}^{k} p(s_i) \tag{5}$$

Theorem 3. *$M \models \mathbf{AF}p$ iff $\exists k\ [\![\, M, \mathbf{AF}p \,]\!]_k$ is valid.*

According to Theorem 3, we need to search for a k that makes the negation of $[\![\, M, \mathbf{AF}p \,]\!]_k$ unsatisfiable. Based on this theorem, we obtain a semi-decision procedure for $M \models \mathbf{AF}p$. The procedure terminates if the liveness property holds. The bound k needed for a proof represents the length of the longest sequence from an initial state without hitting a state where p holds. Based on bounded model checking, we have a semi-decision procedure for $M \models \mathbf{EG}\neg p$, or equivalently, $M \not\models \mathbf{AF}p$. Since we know that either $\mathbf{AF}p$ or $\mathbf{EG}\neg p$ must hold for M, one of the semi-decision procedures must terminate. Combining the two, we obtain a complete decision procedure for liveness.

5.3 Induction

Techniques based on induction can be used to make BMC complete for safety properties [25]. Proving $M \models \mathbf{AG}p$ by induction typically involves finding (manually) a strengthening *inductive invariant*. An inductive invariant is an expression that on the one hand is inductive (i.e., its correctness in previous steps implies its correctness in the current step), and on the other hand it implies the property. Proofs based on inductive invariants have three steps: the base case, the induction step and the strengthening step. Given a bound n, which we refer to as the induction depth, we first prove that the inductive invariant ϕ holds in the first n steps, by checking that Formula (6) is unsatisfiable.

$$\exists s_0, \ldots, s_n.\ I(s_0) \wedge \bigwedge_{i=0}^{n-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{n} \neg\phi(s_i) \tag{6}$$

Next, we prove the induction step, by showing that Formula (7) is unsatisfiable:

$$\exists s_0, \ldots, s_{n+1}.\ \bigwedge_{i=0}^{n} (\phi(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg\phi(s_{n+1}). \tag{7}$$

Finally, we establish that the strengthening inductive invariant implies the property for an arbitrary i:

$$\forall s_i.\ \phi(s_i) \rightarrow p(s_i) \tag{8}$$

If we use the property p as the inductive invariant, the strengthening step holds trivially and the base step is the same as searching for a counterexample to $\mathbf{G}p$.

In a further refinement of Formula (7) suggested by Sheeran etc. [25], paths in M are restricted to contain distinct states. The restriction preserves completeness of bounded model checking for safety properties: if a bad state is reachable, it is reachable via a path with no duplicate states, or, in other words, via a loop-free path. The inductive step is now represented by Formula (9):

$$\exists s_0, \ldots, s_{n+1}.\ \bigwedge_{j=0}^{n} \bigwedge_{k=j+1}^{n+1} (s_j \neq s_k) \wedge \bigwedge_{i=0}^{n} (\phi(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg\phi(s_{n+1}) \tag{9}$$

The restriction to loop-free paths constrains the formula further and hence prunes the search space of the SAT procedure and consequently improves its efficiency. On the other hand, the propositional encoding of distinct state restriction is quadratic with respect to the bound k. When k is large, the restriction may significantly increase the size of the propositional formula. The practical effectiveness of this restriction is to be further studied.

6 Propositional SAT solvers

In this section we briefly outline the principles followed by modern propositional SAT-solvers. Our description follows closely the ones in [30] and [27].

Given a propositional formula f, a SAT solver finds an assignment to the variables of f that satisfy it, if such an assignment exists, or return 'unsatisfiable' otherwise.

Normally SAT solvers accept formulas in Conjunctive Normal Form (CNF), i.e., a conjunction of clauses, each contains a disjunction of literals and negated literals. Thus, to satisfy a CNF formula, the assignment has to satisfy at least one literal in each clause. Every propositional formula can be translated to this form. With a naive translation, the size of the CNF formula can be exponential in the size of the original formula. This problem can be avoided by adding $O(|f|)$ auxiliary Boolean variables, where $|f|$ is the number of sub expressions in f.

Most of the modern SAT-checkers are variations of the well known Davis-Putnam procedure [17] and its improvement by Davis, Loveland and Logemann (known as DPLL) [16]. The procedure is based on a backtracking search algorithm that, at each node in the search tree, decides on an *assignment* (i.e. both a variable and a Boolean value, which determines the next sub tree to be traversed) and computes its immediate implications by iteratively applying the 'unit clause' rule. For example, if the decision is $x_1 = 1$, then the clause $(\neg x_1 \vee x_2)$ immediately implies that $x_2 = 1$. This, in turn, can imply other assignments. Iterated application of the unit clause rule is commonly referred to as Boolean Constraint Propagation (BCP). A common result of BCP is that a clause is found to be unsatisfiable, a case in which the procedure must backtrack and change one of the previous decisions. For example, if the formula also contains the clause $(\neg x_1 \vee \neg x_2)$, then clearly the decision $x_1 = 1$ must be changed, and the implications of the new decision must be re-computed. Note that backtracking implicitly prunes parts of the search tree. If there are *n* unassigned variables in a point of backtracking, then a sub tree of size 2^n is pruned. Pruning is one of the main reasons for the impressive efficiency of these procedures.

```
// Input arg: Current decision level d
// Return value:
//   SAT():     {SAT, UNSAT}
//   Decide():  {DECISION, ALL-DECIDED}
//   Deduce():  {OK, CONFLICT}
//   Diagnose():{SWAP, BACK-TRACK} also calculates β

SAT (d)
   {
l1:    if (Decide (d) == ALL-DECIDED) return SAT;
l2:    while (TRUE) {
l3:       if (Deduce(d) != CONFLICT) {
l4:          if (SAT (d+1) == SAT) return SAT;
l5:          else if (β < d || d == 0)
l6:             { Erase (d); return UNSAT; }
          }
l7:       if (Diagnose (d) == BACK-TRACK) return UNSAT;
       }
   }
```

Fig. 5. Generic backtrack search SAT algorithm

Fig. 5 describes a template that most SAT solvers use. It is a simplified version of the template presented in [30]. At each *decision level d* in the search, a variable assignment $V_d = \{T, F\}$ is selected with the `Decide()` function. If all the variables are already decided (indicated by `ALL-DECIDED`), it implies that a satisfying assignment has been found, and `SAT` returns `SATISFIABLE`. Otherwise, the *implied assignments* are identified with the `Deduce()` function, which corresponds to a straightforward BCP. If this process terminates with no conflict, the procedure is called recursively with a higher decision level. Otherwise, `Diagnose()` analyzes the conflict and decides on the next step. If V_d was assigned only one of the Boolean values, it swaps this value and the deduction process in line l_3 is repeated. If the swapped assignment also fails, it means that V_d is not responsible for the conflict. In this case `Diagnose()` identifies the assignments that led to the conflict and computes the decision level β (β is a global variable that can only be changed by `Diagnose()`) to which SAT() should backtrack to. The procedure will then backtrack $d - \beta$ times, each time `Erase()`-ing the current decision and its implied assignments, in line l_6.

The original Davis-Putnam procedure backtracked one step at a time (i.e. $\beta = d - 1$). Modern SAT checkers include *Non-chronological Backtracking* search strategies (i.e. $\beta = d - j, j \geq 1$), allowing them to skip a large number of irrelevant assignments. The introduction of non-chronological backtracking to SAT solvers in the mid 90's was one of the main breakthroughs that allowed these procedures for the first time to handle instances with tens of thousands of variables (this technique was used previously in general Constraint Solving Problem (CSP) tools. See [30] for more details).

The analysis of conflicts is also used for *learning*. The procedure adds constraints, in the form of new clauses (called *conflict clauses*) that prevent the repetition of bad assignments. This way the search procedure backtracks immediately if such an assignment is repeated. We explain the mechanism of deriving new conflict clauses by following a simplified version of an example given in the above reference.

Example 2. Assume the clause data base includes the clauses listed in Fig. 6(a), the current truth assignment is $\{x_5 = 0\}$, and the current decision assignment is $x_1 = 1$. Then the resulting *implication graph* depicted in Fig. 6 (b) describes the unit clause propagation process implied by this decision assignment.

Each node in this graph corresponds to a variable assignment. The incoming directed edges $(x_1, x_j)...(x_i, x_j)$ labeled by clause c represent the fact that $x_1...x_i, x_j$ are c's literals and that the current value of $x_1, ..., x_i$ implies the value of x_j according to the unit clause rule. Thus, vertices that have no incoming edges correspond to decision assignments while the others correspond to implied assignments. The implication graph in this case ends with a conflict vertex. Indeed the assignment $x_1 = 1$ leads to a conflict in the value of x_4, which implies that either c_3 or c_4 cannot be satisfied. When such a conflict is identified, `Diagnose()` determines those assignments that are directly responsible for the conflict. In the above example these are $\{x_1 = 1, x_5 = 0\}$. The conjunction of these assignments therefore represents a sufficient condition for the conflict to arise. Consequently, the negation of this conjunction must be satisfied if the instance is satisfiable. We can therefore add the new conflict clause $\pi : (\neg x_1 \vee x_5)$ to the clause database, with the hope that it will speed up the search. □

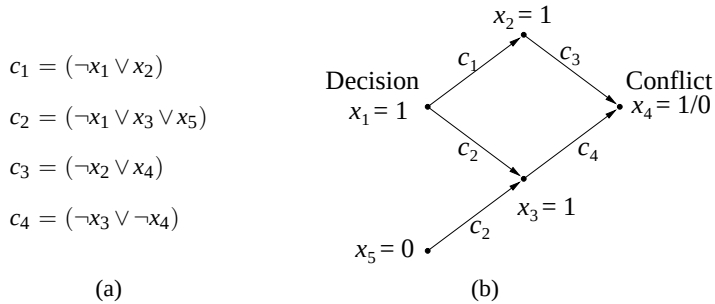

Fig. 6. A clause data base (a) and an implication graph (b) of the assignment $x_1 = 1$ shows how this assignment, together with assignments that were made in earlier decision levels, leads to a conflict.

Another source of constant improvement in these tools is the development of new decision heuristics in `DECIDE()`, i.e. the strategy of picking the next variable and its value. The order can be static, i.e., predetermined by some criterion, or decided dynamically according to the current state of the search. For example, the DLIS strategy [29] picks an assignment that leads to the largest number of satisfied clauses. Although this strategy normally results in a good ordering, it has a very large overhead, since each decision requires a count of the currently unsatisfied clauses that contain each variable or its negation. A recently suggested strategy, called Variable State Independent Decaying Sum (VSIDS) [22], avoids this overhead by ignoring whether the clause is currently satisfiable or not. It counts (once) the number of times each variable appears in the formula, and then updates this number once new conflict clauses are added to the formula. By giving more weight to variables in newly added conflict clauses, it makes the decision *conflict-driven*, i.e. it gives higher priority to solving conflicts that were recently identified. This procedure turned out to be an order of magnitude faster, on average, compared to DLIS.

7 Experiments

Since the introduction of BMC several independent groups published experimental results, comparing BMC to various BDD based symbolic model checkers. In this section we quote some of the experiments conducted by the verification groups at IBM, Intel and Compaq, as well as our own experiments. All of these experiments basically reach the same conclusion: SAT based Bounded Model Checking is typically faster in finding bugs compared to BDDs. The deeper the bug is (i.e. the longer the shortest path leading to it is), the less advantage BMC has. With state of the art SAT solvers and typical hardware designs, it usually cannot reach bugs beyond 80 cycles in a reasonable amount of time, although there are exceptions, as the experiments conducted in Compaq show (see Fig. 10 below). In any case, BMC can solve many of the problems that cannot be solved by BDD based model checkers.

The experiments were conducted with different SAT solvers and compared against different model checkers. The introduction of the SAT solver CHAFF in mid 2001 changed the picture entirely, as on average it is almost an order of magnitude faster than previous SAT solvers. This means that experiments conducted before that time are skewed towards BDDs, compared to what these experiments would reveal today.

The first batch is summarized in Fig. 7. It shows the results of verifying a 16x16 shift and add multiplier, as was first presented in [5]. This is a known hard problem for BDDs. The property is the following: the output of the sequential multiplier is the same as the output of a combinational multiplier applied to the same input words. The property was verified for each of the 16 output bits separately, as shown in the table. For verifying bit i, it is sufficient to set the bound k to $i+1$. This is the reason that the SAT instance becomes harder as the bit index increases. As a BDD model checker, we used B. Yang's version of SMV, which is denoted in the table as SMV_2. The variable ordering for SMV was chosen manually such that the bits of registers are interleaved. Dynamic reordering did not improve these results.

bit	k	SMV_2	MB	PROVER	MB
0	1	25	79	<1	1
1	2	25	79	<1	1
2	3	26	80	<1	1
3	4	27	82	1	2
4	5	33	92	1	2
5	6	67	102	1	2
6	7	258	172	2	2
7	8	1741	492	7	3
8	9		>1GB	29	3
9	10			58	3
10	11			91	3
11	12			125	3
12	13			156	4
13	14			186	4
14	15			226	4
15	16			183	5

Fig. 7. Results in seconds and Mega-Byte of memory when verifying a 16x16 bit sequential shift and add multiplier with overflow flag and 16 output bits.

A second batch of comparisons was published in [27]. It presents a comparison between RULEBASE, IBM's BDD based symbolic model checker, and several SAT solvers, when applied to 13 hardware designs with known bugs. The columns RULEBASE_1 and RULEBASE_2 represent results achieved by RULEBASE under two different configurations. The first is the default configuration, with dynamic reordering. The second is the same configuration without reordering, but the initial order is taken from the order that was calculated with RULEBASE_1. These two configurations represent a typical scenario of Model Checking with RULEBASE. Each time reordering is activated, the

initial order is potentially improved and saved in a special order file for future runs. The column 'GRASP' contains results of solving the corresponding BMC formulas with the SAT solver GRASP. The following column, 'GRASP (tuned)', contains results of solving the same instances with a version of GRASP that is tuned for BMC, as explained in the above reference. The last column was not part of the original presentation in [27]; rather it was added for this article. It contains results achieved by CHAFF on the same benchmarks, without any special tuning (CHAFF was released after the above reference was published). The fact that CHAFF can solve all instances, while GRASP, which was considered as the state of the art solver before CHAFF, cannot solve it even with special tuning, demonstrates the great progress of SAT solvers and the influence of this progress on BMC.

Model	k	RULEBASE$_1$	RULEBASE$_2$	GRASP	GRASP (tuned)	CHAFF
Design 1	18	7	6	282	3	2.2
Design 2	5	70	8	1.1	0.8	< 1
Design 3	14	597	375	76	3	< 1
Design 4	24	690	261	510	12	3.7
Design 5	12	803	184	24	2	< 1
Design 6	22	*	356	*	18	12.2
Design 7	9	*	2671	10	2	< 1
Design 8	35	*	*	6317	20	85
Design 9	38	*	*	9035	25	131.6
Design 10	31	*	*	*	312	380.5
Design 11	32	152	60	*	*	34.7
Design 12	31	1419	1126	*	*	194.3
Design 13	14	*	3626	*	*	9.8

Fig. 8. The IBM® benchmark: verifying various hardware designs with an in-house BDD model checker (RULEBASE) and the SAT solver GRASP with and without special tuning. The last column presents the results achieved with the newer SAT solver CHAFF on the same benchmark examples. Results are given in seconds.

The next benchmark examples was published in [14] by the formal methods group of Intel. They compared the run time of their BDD model checker FORECAST and their bounded model checker THUNDER (based on a SAT solver called SIMO) when applied to 17 different circuit designs. The table in Fig. 9 summarizes the results of their comparison when the two tools are run under their default configuration[4].

Finally, Compaq published another batch of results obtained with industrial examples [6]. They used bounded model checking with the PROVER SAT solver for finding bugs in the memory system of an advanced Alpha microprocessor. Their conclusion was similar to the previous published comparative research: SAT based bounded model

[4] Other tables in the above reference show that with manual intervention in choosing the variable order the results can change in favor of FORECAST.

Model	k	FORECAST (BDD)	THUNDER (SAT)
Circuit 1	5	114	2.4
Circuit 2	7	2	0.8
Circuit 3	7	106	2
Circuit 4	11	6189	1.9
Circuit 5	11	4196	10
Circuit 6	10	2354	5.5
Circuit 7	20	2795	236
Circuit 8	28	*	45.6
Circuit 9	28	*	39.9
Circuit 10	8	2487	5
Circuit 11	8	2940	5
Circuit 12	10	5524	378
Circuit 13	37	*	195.1
Circuit 14	41	*	*
Circuit 15	12	*	1070
Circuit 16	40	*	*
Circuit 17	60	*	*

Fig. 9. The Intel® benchmark: verifying various circuit designs with an in-house BDD model checker (FORECAST) and an in-house SAT solver (THUNDER). Results are given in seconds.

checking can solve in a short amount of time examples that cannot be solved with a BDD based model checker. Their results are summarized in Fig. 10.

k	SMV	PROVER
25	62280	85
26	32940	19
34	11290	586
38	18600	39
53	54360	1995
56	44640	2337
76	27130	619
144	44550	10820

Fig. 10. The Compaq® benchmark: verifying an Alpha microprocessor with BDDs (SMV) and SAT (PROVER). Results are given in seconds.

8 Related Work and Conclusions

Verification techniques based on satisfiability checking have been used since the early 90's by G. Stalmarck and his company Prover Technologies [31]. The method is based on the patented SAT solver PROVER [26], that is very effective in tackling structured problems that arise from real-world designs. The work in [31] focuses on checking correctness of designs by means of inductive reasoning, as was explained in section 5.3. Impressive results have been achieved in terms of integration of this technique within the development process in several domains (see e.g. [7]).

The initial successes of BMC drew attention from the verification community. It has been introduced in several model checkers (e.g. NuSMV [10]), and a number of advances have been achieved in several directions, which we briefly describe now.

In [27], Strichman showed that it is possible to tune SAT solvers by exploiting the structure of the problem being encoded in order to increase efficiency. Notable contributions in [27] and [28] are the use of problem-dependent variable ordering and splitting heuristics in the SAT solver, pruning the search space by exploiting the regular structure of BMC formulas, reusing learned information between the various SAT instances and more. These improvements were the basis for the tuned SAT solver presented in Fig. 8. The work in [32] pushes this idea further. It relies on an incremental SAT solver, rather than on generating a new SAT instance for each attempted bound. At each step, they add and remove clauses from a single SAT instance, and this way retain the learned information from the previous instances, as was independently suggested in [28].

A related development was the extension of Bounded Model Checking to Timed Systems [2]. For this purpose they use MATHSAT [1], a SAT solver extended to deal with linear constraints over real variables. The encoding style extends the encoding for the untimed case, and uses constraints over real variables to represent the aspects related to time.

The success of SAT in solving large problems led several groups to combine SAT in various ways with other techniques used in verification, not just as part of BMC. We will mention here two of these works. McMillan [21] recently introduced a SAT-based *unbounded* CTL model checker. It is based on an quantifier elimination procedure similar to [23, 24]. While the top level algorithm is basically the same as used in BDD-based CTL model checking, sets of states are represented as CNF formulas rather than with BDDs. This required a modification of the SAT solver in order to be able to perform the key operation of quantifier elimination. His experimental results show that this technique can compete with BDD based model checkers and in some cases outperform it. Compared to BMC, it has the obvious advantage of reaching a fixpoint after $rd(M)$ steps, rather than after $rdr(M)$ steps (see section 5.1), which is only an over approximation of $rd(M)$. Currently there is no available data comparing this technique to BMC.

SAT-based techniques have also been used in the framework of abstraction / refinement [12]. While a BDD based model checker is used to prove the abstract model, SAT solvers are used to check whether the counterexamples constructed in the abstract space are real or spurious, and also to derive a refinement to the abstraction being applied. This procedure relies on the speed of SAT to check whether a given trace (i.e. with a known length, as in BMC) is real. On the other hand it enjoys the completeness guaranteed by using BDD based model checkers.

A recently published work by Baumgartener et al. [3] holds a large promise for making BMC complete for a large class of hardware designs. They perform a structural analysis of the design in order to derive an over approximation of the reachability diameter, thus achieving completeness. The experiments show that the reachability diameter of realistic designs can be reached, and hence the property can be proved. This work was published only recently, and its affect is not yet clear. The authors of [3] showed that for a large class of netlists, it is possible to find smaller reachability diameters than those that are defined by Formula (4). This requires a fairly simple analysis of the netlist structure, identifying frequently occurring components like memory registers, queue registers, etc., and identifying its Strongly Connected Components (SCC). The overall reachability diameter is then defined recursively on the reachability diameters of its individual SCCs. Their experiments showed that many netlists have reachability diameters as small as 20, which means that they can be easily proved with BMC. It is perhaps too early to judge to what degree this improvement will make BMC viable for verification, rather than for falsification alone.

Despite its recent introduction, Bounded Model Checking is now widely accepted as an effective technique that complements BDD-based model checking. A typical methodology applied in the industry today is to use both BMC and BDD based model checkers as complementary methods. In some cases both tools are run in parallel, and the first tool that finds a solution, terminates the other process. In other cases BMC is used first to find quickly the more shallow bugs, and when this becomes too hard, an attempt to prove that the property is correct is being made with a BDD based tool. In any case, it is clear that together with the advancements in the more traditional BDD based symbolic model checkers, formal verification of finite models has made a big step forward in the last few years.

References

1. G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A sat based approach for solving formulas over boolean and linear mathematical propositions. In *18th Int. Conference of Automated Deduction (CADE'02)*, LNAI, Copenhagen, July 2002. Springer-Verlag.
2. G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. Bounded model checking for timed systems. In *22nd Joint International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2002)*, Lect. Notes in Comp. Sci., Houston, TX, November 2002. Springer-Verlag.
3. J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 151–165, 2002.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS. Springer-Verlag, 1999.
5. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, , and Y. Zhu. Symbolic model checking using sat procedures instead of BDDs. In *Design Automation Conference (DAC'99)*, 1999.
6. P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In G. Berry, H. Comon, and A. Finkel, editors, *Proc.* 12^{th} *Intl. Conference on Computer Aided Verification (CAV'01)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2001.

7. A. Boralv and G. Stalmarck. Prover technology in railways. In *Industrial-Strength Formal Methods*. Academic Press, 1998.
8. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.
9. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
10. A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 359–364, 2002.
11. E. M. Clarke and A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *In Logic of Programs: Workshop, Yorktown Heights*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.
12. E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine lea rning techniques. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 265–279, 2002.
13. E. M. Clarke, O.Grumberg, and D.Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
14. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proc.* 12^{th} *Intl. Conference on Computer Aided Verification (CAV'01)*, Lect. Notes in Comp. Sci., pages 436–453, 2001.
15. O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. IEEE International Conference on Computer-Aided Design*, 1990.
16. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
17. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
18. H. Kautz and B. Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proc. AAAI'96*, Portland, OR, 1996.
19. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Poceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.
20. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.
21. K. L. McMillan. Applying sat methods in unbounded symbolic model checking. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 250–264, 2002.
22. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference 2001 (DAC'01)*, 2001.
23. D. Plaisted. Method for design verification of hardware and non-hardware systems. United States Patent, 6,131,078, October, 2000.
24. D. Plaisted, A. Biere, and Y. Zhu. A satisfiability procedure for quantified boolean formulae. *Discrete Applied Mathematics*, 2002. accepted for publication.
25. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a sat-solver. In Hunt and Johnson, editors, *Proc. Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2000)*, 2000.
26. M. Sheeran and G. Stalmarck. A tutorial on stalmarck's method. *Formal Methods in System Design*, 16(1), January 2000.
27. O. Shtrichman. Tuning SAT checkers for bounded model checking. In E.A. Emerson and A.P. Sistla, editors, *Proc.* 12^{th} *Intl. Conference on Computer Aided Verification (CAV'00)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2000.

28. O. Shtrichman. Prunning techniques for the SAT-based bounded model checking problem. In *proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Edinburgh, September 2001.
29. J. P. M. Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *9th Portuguese Conference on Artificial Intelligence (EPIA)*, 1999.
30. J. P. M. Silva and K.A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical Report TR-CSE-292996, Univerisity of Michigen, 1996.
31. G. Stålmarck and M. Säflund. Modelling and Verifying Systems and Software in Propositional Logic. *Ifac SAFECOMP'90*, 1990.
32. J. Whittemore, J. Kim, and K.A. Sakallah. Satire: A new incremental satisfiability engine. In *Design Automation Conference (DAC'01)*, pages 542–545, 2001.