# About Bounded Model Checking and Interpolation

**Sebastian Henningsen[1] and Manuel Hoffmann[1]**

1 **Technische Universität Kaiserslautern**
**Gottlieb-Daimler-Straße 48, 67663 Kaiserslautern, Germany**
`{s_henningsen09,m_hoffmann09}@cs.uni-kl.de`

## Abstract

Model checking has become a rich and versatile methodology for the property verification of hardware and software systems. However, most model checking problems are computationally hard and applicability in practice is limited by the state explosion problem. Instead of exhaustive verification, bounded model checking (BMC) aims at the falsification of properties (i.e. testing/bug hunting) by unrolling the transition relation of a system finitely many times. Hence, BMC only inspects a small fragment of the total state space where, as it is claimed in the literature, most bugs are likely to happen.

In addition to solely falsification, there are several extensions to BMC that can prove certain properties, for example interpolation and k-induction. Moreover, BMC-based approaches have been proposed for the verification of concurrent and C programs. In contrast to unbounded symbolic model checking using BDDs, BMC problems are expressed by satisfiability problems which can be solved using a SAT or SMT solver. Given the recent advances in SAT and SMT solvers, this oftentimes leads to a significant performance benefit over traditional approaches.

**1998 ACM Subject Classification** D.2.4 Model checking

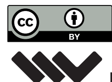
**Keywords and phrases** Bounded model checking, Interpolation

## 1 Introduction

Ever since it's beginnings in the seminal works of Clarke and Emerson [4] and Queille and Sifakis [17], Model Checking (or Property Checking) has undergone a tremendous development and has received, in recent years, lots of attention from the industry. The problem model checking refers to is surprisingly simple: Given a model of a system, check algorithmically whether the model fulfills a given specification. Oftentimes, the specification is given in temporal logic (e.g. LTL or CTL) whereas the model can be any formal description of a software or hardware system.

In this work we will cover the ideas, related work and applications of bounded model checking (BMC) and Craig's interpolants – two subfields of symbolic model checking which are linked tightly. In contrast to unbounded model checking, BMC aims at the falsification of safety properties, instead of verification. Basically speaking, BMC searches for counter-examples by unrolling the transition relation of the system model finitely often, hence the term "bounded". Moreover, this principle has been extended by various techniques, a more prominent example being interpolants, which allows for unbounded model checking by using BMC.

The remainder of this work is organized as follows: In this section a brief introduction to model checking and relevant concepts is given. Section two addresses the use of SAT in contrast to BDDs in symbolic model checking. After this basic setup, section three introduces bound model checking and discusses some of its properties. Section four introduces the concept of interpolants and enriches the previously presented techniques. Last but not least, a conclusion is given in section five.


© Sebastian Henningsen, Manuel Hoffmann;
licensed under Creative Commons License CC-BY

## 1.1 Notation and preliminaries

### 1.1.1 Description of Models

Model checking answers the question whether a system $\mathcal{K}$ satisfies a property $\varphi$, or formally: $\mathcal{K} \models \varphi$, pronounced "$\mathcal{K}$ is a model for $\varphi$" – hence the name. Typical formal representations of $\mathcal{K}$ are Kripke structures [19], Petri Nets [17] or timed automata [23]. All of these representations consist of states and transitions between them that modeling steps of the original system.

When we want to examine the properties of a system (e.g. a software system or a circuit description), we have to describe the *behavior* of the system in terms of states and transitions, and the *specification* the system should fulfill. The former is given by the implementation (the source code or the circuit layout), the latter induced by the requirements that are to be met by the system.

To exemplify, let the system be modeled by a Kripke structure. A Kripke structure consists of finitely many states, the set of initial states, a transition relation between these states and a labeling from states to a vector of variables [19]. In the model-checking scenario, the states could represent program states, the transition relation possible control flows through the program and the labeling could describe the memory contents in this state.

Figure 1 shows an exemplary Kripke structure with states $s_1, s_2, s_3$ where $\{s_1\}$ is the set of initial states. The transitions possible in this Kripke structure are for example $s_1$ to $s_2$ or $s_3$ to $s_3$, but not from $s_3$ to any other state. The labeling here indicates that in $s_1$ are $p$ and $q$ true and in $s_2$ ($s_3$) is only $p$ ($q$) true.

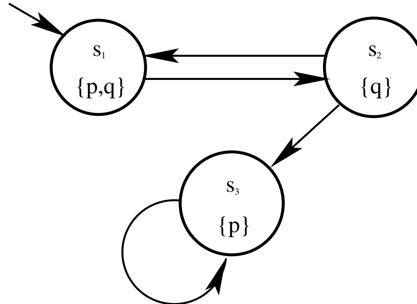



**Figure 1** Simple Kripke structure [24]

### 1.1.2 Formulating of Specifications

In order to express the specification, there are different temporal logics in use, like computation tree logic (CTL) and linear time logic (LTL), where LTL is favored by SAT-based model checking [1]. LTL consists of standard propositional logic with temporal operators like finally **F**, globally **G** and next **X**. The semantics of these operators are defined for paths through the transition system. A path $\pi$ through the transition system is a sequence of states $s_0, s_1, s_2, \ldots$ where an occurrence of $s_i$ and $s_{i+1}$ implies that the system's transition relation includes $(s_i, s_{i+1})$. The formula $p$ ($\neg p$) holds on a path $\pi$, if $p$ is (not) contained in the label of the first state of $\pi$. $\mathbf{G}\,p$ holds if $p$ is included in every state of the path, while $\mathbf{F}\,p$ holds if there is one state labeled with $p$ on the path. $\mathbf{X}\,p$ means, that p has to hold on the next state of $\pi$. Such a formula is called *valid* in a Kripke structure if it holds for every path starting in an initial state.

Model Checking then means that a proposition about the system's behavior is checked. For example, one can be interested in the question whether a program can reach a state where the condition $q$ holds or if the condition $p \to q$ holds along every path through the structure. This question is typically formulated in a temporal logics expression $\varphi$ which serves as input for the Model Checker. It evaluates all the states of the transition system where $\varphi$ holds and finally checks whether the initial states belong to them. The first example could be encoded with the LTL formula $\mathbf{F}\, q$ (valid in figure 1), for the second example $\mathbf{G}(p \to q)$ would be an adequate LTL statement (not valid in figure 1).

## 2 Bounded Model Checking

Traditionally, model checking was done using symbolic representation with binary decision diagrams (BDDs) as they made it possible to handle mode than $10^{20}$ states, and could handle systems with hundreds of state variables, which is not feasible with explicit state traversal [3]. But the size of the BDD and thus the execution time of the running model checker is heavily influenced by the variable ordering in the BDD.

SAT-based approaches on the other hand do not necessarily rely on canonical forms and therefore have no problem with growth of needed space like BDDs, and they can easily handle thousands of variables. In the original work of Biere, Clarke et al., they presented the tool *BMC* based on bounded model checking, which makes use of SAT based counter example finding.

### 2.1 Bounding

Model checking specifications can be divided into universal model checking problems (as all paths through the transition system have to be considered) and existential model checking problems. For solving the former, the negation can be used to check for counter examples. Bounded model checking means now "to consider only a finite prefix of a path" [3] and so find either a counter example or a witness for the problem. We will refer to the length of this prefix by $k$.

Obviously, if in a Kripke structure $\mathcal{K}$ there is a state with label $p$ in distance $l \leq k$ of an initial state and we want to check whether $\mathcal{K} \models \mathbf{G}\,\neg p$ holds, we can find a counter example by searching for the inverse specification $\mathcal{K} \models \mathbf{F}\, p$ which has only to hold for one path.

In order to generate the propositional formula for the SAT solver, the transition relation is unroled $k$ times such only states are included that are reachable by such a path of length $k$. Although only a fixed number of steps in a path is checked, there are patterns which make it possible to reason about infinite paths. One possibility is, when there is a loop from state number $l$ back to $i$ with $i < l \leq k$, then also global properties can be shown for an infinte path going through this loop. With these considerations, the model checking problem is rephrased to the following SAT problem:

$$\underbrace{I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})}_{\text{unfolding the transition relation}} \quad \wedge \left( \underbrace{[\![\varphi]\!]_k^0}_{\text{no loops}} \quad \vee \quad \underbrace{\bigvee_{l=0}^{k} \lambda_l \wedge \ {}_l[\![\varphi]\!]_k^0}_{\text{loops of length } l} \right)$$

Where $[\![\varphi]\!]_k^0$ is the translation of $\varphi$ for k steps starting in state 0, $\lambda_l$ implies the presence of a loop from state $s_l$ to a previous state and ${}_l[\![\varphi]\!]_k^0$ is the translation of $\varphi$ with a loop start $l$. The details for these translations can be found in [1].

```
1  x=x+y;
2  if(x!=1)
3    x=2;
4  else
5    x++;
6
7  assert(x<=3);
```

```
1  x1=x0+y0;
2  if(x1!=1)
3    x2=2;
4  else
5    x3=x1+1;
6  x4=(x1!=1)?x2:x3;
7  assert(x1<=3);
```

$$
\begin{aligned}
C := {} & x_1 = x_0 + y_0 \\
& \wedge\ x_2 = 2 \\
& \wedge\ x_3 = x_1 + 1 \\
& \wedge\ x_4 = (x_1 \neq 1) ? x_2 : x_3 \\
P := {} & x_4 \leq 3
\end{aligned}
$$

**Figure 2** From C code (left) over intermediate-code in SSA (middle) to bit-vector equations, as illustrated in [5].

# 3 Applications of BMC

In this section we look into two examples of model checking for multithreaded C programs.

## 3.1 Rabinovitz and Grumberg's BMC

In 2004, Clarke et al. presented *CBMC*, a tool that can show safety properties of low-level C programs and uses bounded model checking [5]. CBMC can handle pointers and arrays, loops and function calls. In a preprocessing step it unfolds `while` loops and recursive function calls $n$ times, for a "big enough $n$". After preprocessing the remaining program consists of `if` instructions, assignments, assertions, labels and `goto`-forward instructions. Lastly, the program is transformed into single assignment form (SSA). From this the constraints $C$ and properties $P$ are extracted and $C \wedge \neg P$ is fed into a SAT solver in order to find a counter example.

Figure 2 shows an exemplary transformation from C code to SSA-form which is pretty straight forward. Remarkably the state of variable `x` after the `if-else`-block is captured by the conditional assignment in line 6.

### 3.1.1 The Method of TCBMC

In 2005, Rabinovitz and Grumberg presented *TCBMC* (for "Threaded-C"), an extension of CBMC that allows checking of programs which run with different threads and have thread-local and global variables [18]. As they assume, that bug patterns appear mostly in few context switches, they bound the number of allowed context switches to a certain number $n$.

Intuitively, the concurrent execution of a threaded C program is one of the arbitrary interleavings of the threads. We even can decrease the number of interesting interleavings, as for the program only context switches that occur before accesses to global variables can make a difference in the result (valuation for variables, heap state) of the program. But C statements themselves are not necessarily atomic, as variable assignments may involve load and store instructions. So, TCBMC starts by translating assignments involving global variables like `x = y + z;` into the following form: `l1 = y; l2 = y; z = l1 + l2;`.

The next step is, that every thread is put into CBMC which returns the list of constraints, which Rabinovitz and Grumberg interpret as program where "each constraint is an assignment and each variable is assigned only once". These variables are copies of the variables of the original program.

For every line `l` of thread `t` of this produced program, they introduced a variable

```
1 atomic {
2   assume(*mutex == U);
3   *mutex = L;
4 }
```

```
1 atomic {
2   assert(*mutex == L);
3   *mutex = U;
4 }
```

**Figure 3** C code for locking and unlocking mutexes with TCBMC [5].

`thread`$_t$`_cs(l)` which indicates the number of context switches that occured before this line was executed. All lines of code with the same value of `thread`$_t$`_s` belong to the same context switch block, of which at most $n$ exist. Also, for each globally visible variable $x$ of the program, they add $n$ variables $x_i, i = 1 \dots n$, where $x_i$ indicates the value of $x$ at the end of context switch block $i$.

In order to support synchronization primitives, they started by modeling atomic sections. As these are not natively supported by C, they introduced new constraints enforcing the value of `thread`$_t$`_cs(l)` staying constant for every `l` in the atmoic block. Mutexes can be modeled on top of this by the code shown in figure 3.

### 3.1.2 Using TCBMC to find Races and Deadlocks

A race occurs, when two threads access a global variable and at least one of these accesses is a write operation. TCBMC supports race detection by adding a write-flag to every variable $x$ which is checked to be false before every access and set true for one instruction after every write and one instruction later set to false again. Checking, writing and setting to true happens in one atomic block, however, setting to false again is subject to a context switch. This way, when a context switch occurs directly after the write, the assertion of the next read/write to $x$ fails in case of a race.

For deadlock detection, they focused on local and global deadlocks. Global deadlocks happen, when all threads are in a waiting state, while local deadlocks are characterized by only some threads forming a waiting cycle. Rabinovitz and Grumberg described roughly the global case which works as follows: There is a global counter *threads_in_wait* which is increased if a locking operation on a locked mutex has failed. Then the context switch to the next operation happens and it is asserted that *threads_in_wait* is strictly smaller than the number of threads. But if *threads_in_wait* equals the number of threads then a global deadlock is found.

## 3.2 Corderio and Fischer's BMC

Corderio and Fischer also went for validation of multi-threaded C code in their 2011 paper where they described “a comprehensive SMT-based BMC procedure” [6].

Like in [18] they restrict the number of context switches by considering only switches at accesses to global variables. Their approach for describing reachable states relies on the construction and exploration of a *reachability tree* where each node is labeled with the running thread, a context switch number, the current state, the current location of all threads and control flow guards. The latter makes sure, that the branch condition holds after a taken branch, does not hold after a not-taken branch and that after an assertion, the statement about the variable does hold.

Figure 4 shows the control-flow graph of a C program that consists of variable assignments (L3, L5, L9, L10, L12), conditional branches (L5, L11), a declaration (L8) and instructions to manage the lifetime of threads (L2, L6, L7, L13). In the original C program there are

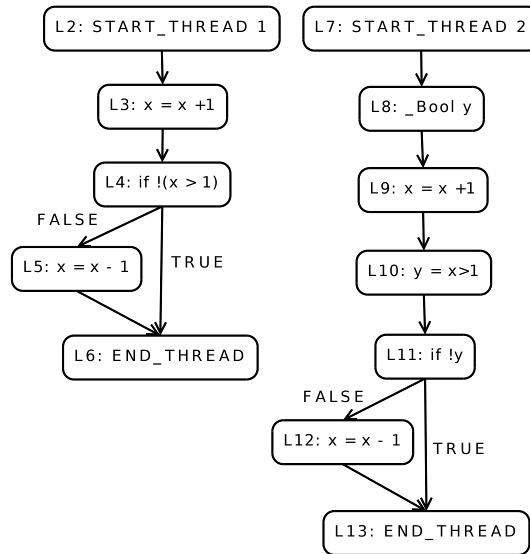



**Figure 4** Control-flow graph of a simple two threaded C program as presented in [6]

also instructions to start the two threads independently, and after the end of both threads asserting $x == 1$. A fragment of the reachability tree for this program depicted in figure 5 illustrates the algorithm, starting with thread 0, the context switch number 0, the state where $x = 0$ and $y$ is defined but not set, as well as the control-flow guards for starting the threads at L2 and L7. Also the main thread which is not shown here is active on L16.

The algorithm for generating the reachability tree works by consuming the current node $e$ of the tree and producing its children. For producing the children there are the following possibilities, depending on the instruction $I$ referred to by the current location of the active thread, the state $s$. It works following these rules, whose effects are demonstrated with the fragment in figure 5[1]:

- If $I$ is an assignment, a child node with state $s'$ which reflects the assignment and the updated current location of the active thread is added. Also for each other enabled thread, a child node with $s'$ and changed active thread marker is added to the tree.
  In the example, node $v_1$ is inserted as child of $v_0$ where $t_1$ is now active, we have 1 context switch seen so far, in the state x=1 holds and y remains undefined, and the active instruction for thread 1 is now L3.
- If $I$ is *skip*, the state remains the same and the current location of the active thread is updated.
- If $I$ is a conditional branch, then two nodes are created, one for taking the branch and one for not taking it. The (negated) branching condition is added to the thread's guards and the active instruction marker is updated.
  In from $v_5$ to $v_6$ there is a context switch back to the branch L4, it's condition can be evaluated to false, as x=2 was true in $v_5$, and the instruction L5 can be executed. Also as the condition was false, the guard for thread 1 is now false.

[1] This fragment seems to have several typos. As it still provides an intuition, we decided to keep it in this work.

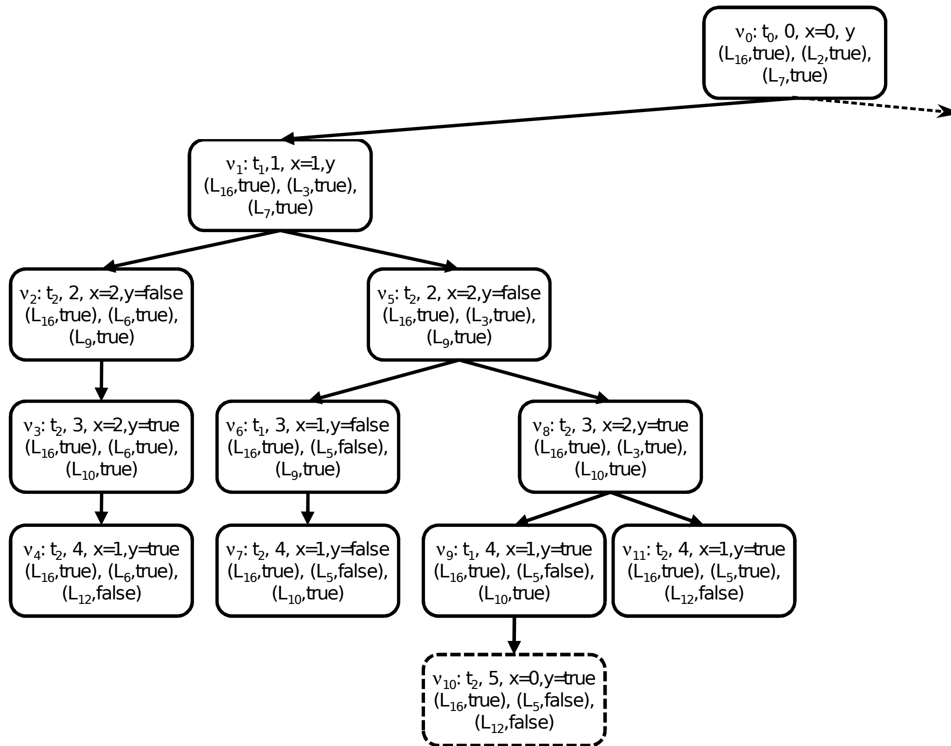



**Figure 5** Fragment of the reachability tree of the same program where the state marked with a dashed line represents a program location that violates an assertion according to [6]

.

- If $I$ is an assertion, then the asserted predicate is used to generate a verification condition to check its validity.
  Node $v_1 0$ has a state where x=0 holds, but in the same time the verification condition forces x=1 to hold.
- If $I$ is *start_thread*, the created node contains the new thread and its location is set to the thread's initial location. Also the thread starts with the guards of the active thread. If $I$ is however a *join_thread* instruction, the current thread's location marker is only incremented, if the joined thread's exit marker is set (done on *end_thread*).

The reachability tree is constructed in a depth-first manner where when a leaf node is found (e.g. forced by hitting the context switch threshold) the path is used to formulate a SAT-problem:

$$I(s_0) \land R(s_0, s_1) \land R(s_1, s_2) \land \ldots \land R(s_{k-1}, s_k) \land \neg\varphi \tag{1}$$

This formula is composed of the path constraints describing the root ($I(s_0)$) and each transition towards the leaf ($R(s_i, s_{i+1})$) and the negated verification condition ($\varphi$) derived from the assertions observed on the path. With an according theory (e.g. linear arithmetics) this formula is fed into an SMT solver. If it finds that this formula is satisfiable, it returns a satisfying assignment which can be used to construct the concrete trace leading to the verification violation. Also this formula shows the relation to the original BMC idea by Biere et al. [3].

# 4 Interpolants

Bounded model checking in its original form can only be used for falsification, unless an upper bound on the depth of the state space is known. The computation of such an upper bound may be expensive or even impossible and approximation schemes (like the longest simple path, i.e. a path which contains every node at most once) are often poor in quality.

Interpolants provide an efficient methodology to extend bounded model checking techniques to the unbounded case, based solely on a SAT-solver. Moreover, interpolants can also be used for predicate abstraction and predicate refinement [16] as well as theorem proving [14]. Related techniques for verification using BMC include k-induction [8, 22], which is somewhat similar to the interpolation approach discussed below. For a given safety property $p$ BMC-like instances are set up such that in the end either a counterexample or a proof that $p$ is inductive w.r.t. the transition relation is found. In this particular sense, inductive means that $p$ holds in every transition step.

## 4.1 Introduction to Interpolants

Originating from model theory, interpolants have various applications in (bounded) model checking, probably the most prominent being the over approximation of the reachable states from failed BMC proofs.

They were first introduced in the work by Craig [7] where the existence of “intermediary formulas” (as they were called in his work) for first-order predicate logic is proven. Given this result, also referred to as Craig’s theorem, interpolants can be defined accordingly:

▶ Definition 1. Assume $A \to B$ holds in some logic. An interpolant (sometimes Craig interpolant) is a formula $I$ such that:

1. $A \to I$ and $I \to B$ are valid and
2. every non-logical symbol in $I$ occurs in both $A$ and $B$

where non-logical symbols are variables, free function symbols etc.

Obviously, if such an $I$ exists, then $A \to B$ is valid, whereas the converse (i.e. the existence of $I$ if $A \to B$ is valid) holds by Craig’s theorem.

In their application in model checking, interpolants are frequently defined “in reverse”: If $A \wedge B$ is unsatisfiable, then there is in interpolant $I$ such that $A \to I$ is valid and $I$ and $B$ are unsatisfiable, i.e. $I, B \to \bot$ and $I$ contains only global variables [1]. Various application domains of interpolants are covered in [15], for example BMC, predicate abstraction and theorem proving, to name a few.

There are numerous possibilities to compute interpolants. For example given a refutation proof for $A \wedge B$, (e.g. by resolution in propositional logic), an interpolant for $A$ with respect to $B$ can be derived in linear time. This procedure is covered in [1] where a calculus for the annotation of resolution proofs of $A \wedge B$ is introduced which yields a valid interpolant when the empty clause has been derived.

## 4.2 Application in Bounded Model Checking

The key advantage of bounded model checking, as the name already suggests, is the boundedness which allows an encoding of a LTL formula into a purely propositional one. There are different approaches to this transformation (even a linear one), some of which are covered in [1]. Please note that, unless otherwise stated, the following elaborations only hold for *finite* state systems.

A bounded model checking problem consists of formulas characterizing the initial states $I$, the transition relation $T$ and final states (or bad states) $F$ which are instantiated for each time frame $0 \ldots k$. The states are encoded by formulas $s_i, i = 0, \ldots, k$ which are vectors of all variables at time $i$. Please refer to [13] for a rigorous introduction.

$$BMC_j^k = I(s_0) \wedge \left( \bigwedge_{0 \leq i \leq k} T(s_i, s_{i+1}) \right) \wedge \left( \bigvee_{j \leq i \leq k} F(s_i) \right) \tag{2}$$

Thus, this formula expresses the fact that a final state is reachable from the initial states within at least $j$ but at most $k$ steps. Furthermore, in symbolic model checking one is interested in the reachability of certain states without a bound on the steps taken. This query can be realized by symbolic reachability analysis which basically computes the least fixpoint of transition relation applied to the initial states (also called the *image* of the initial states). However, this image computation is oftentimes not very efficient and may be more restricting then needed to prove the unreachability of $F$. Hence, an over-approximation of the reachable states which is tight enough to capture the reachability of $F$ but loose enough to make the computation more efficient would be desirable.

By partitioning formula 2 into two parts $A$ and $B$, we are able to obtain such an over-approximation by using the interpolant of $A \wedge B$. Part $A$ is representing the initial states and one transition step and part $B$ is consisting of the remaining $k-1$ steps and final state constraints, see figure 6 (taken from [13]) for an illustration.

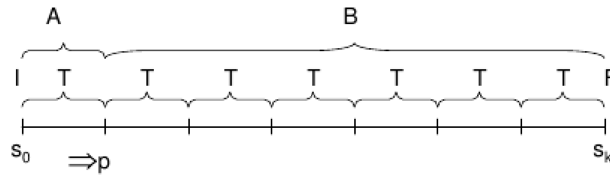



**Figure 6** Partition of a bounded model checking encoding

Further steps depend on the satisfiability of the propositional formula $A \wedge B$: If it is satisfiable, then there is a path from the initial states in up to $k$ steps to the final states, we have thus found a counter example to the property of interest (note that this is “normal” BMC). Otherwise, meaning that there are no counterexamples of length $k$, an interpolant $P$ can be derived from the unsatisfiability proof with the following properties:

1. $A \rightarrow P$, in other words $P$ is an over-approximation of the (one step) reachable states from the initial states
2. $B, P \rightarrow \bot$, meaning that every state which satisfies $P$ cannot reach a bad state in $k-1$ steps

By iterating this procedure, we obtain an over-approximation $R$ of the reachable states from $I$. If $R$ is inconsistent with $F$, then we know $F$ not to be reachable, otherwise no guarantees can be given (since the over-approximation may have been too coarse). In the latter case, the steps above are repeated with a larger $k$. For finite-state systems it has been shown that with increasing $k$ either a counterexample is found or the approximate image operator becomes an adequate image operator if $k$ is greater than the diameter of the state space.

Note that the informal procedure above was first published by [13] and is restricted to safety properties. However, Biere and Schupan [2, 20, 21] proposed a linear “liveness to safety” translation which extends the applicability of the interpolation-based procedure by McMillan.

A practical evaluation of interpolation based model checking can be found in [15] where the procedure was tested against a SMV model checker which performed an exact reachability analysis and a proof based abstraction scheme. The results show that the use of an over-approximation instead of exact analysis pays off greatly in terms of performance.

## 4.3 Application in Predicate Abstraction

The size of the state space (also referred to as state-space explosion) is a major constraint of the effectiveness of model checking algorithms, especially in software verification where there are usually infinitely many states. Therefore, abstraction techniques were introduced which are a methodology to reduce the state space of a system by mapping a set of states to an abstract state – similar to equivalence classes in mathematics. Furthermore, the abstraction should either be exact, i.e. the abstract system has the same behaviors as the original one, or an over-approximation of the possible behaviors. In simplified terms, abstraction removes all components from the model that are irrelevant for checking the property of interest.

There are different types of abstractions, e.g. localization reduction and predicate abstraction [12] – we will only focus on the latter.

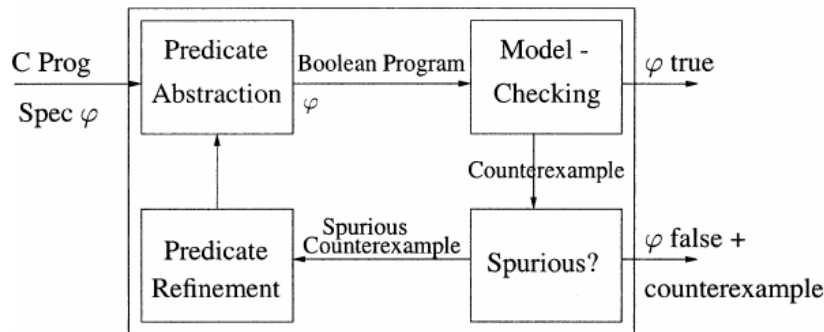



**Figure 7** Schematic illustration of CEGAR

Predicate abstraction, in its automated form first introduced in [9], is mainly used in program abstraction. The key idea is to abstract the data of a program by characterizing the relationships between variables as predicates. An example is illustrated in figure 8 where, instead of storing each possible value of $x$ and $y$, only the predicates $p_1(s) = s.x < s.y$ and $p_2(s) = (s.y = 0)$ are kept (where $s$ refers to the state). Hence, the original program is translated into an abstract one with only Boolean variables representing the different predicates. Reachability analysis of bad states is then carried out on such an abstract program, which is oftentimes an over-approximation of the original behaviors. Hence, in case of a counterexample, one has to check whether it corresponds to a counterexample in the original program: If not (called spurious counter example), the predicate abstraction is refined and the process iterated. This iterative process of predicate refinement is called *Counterexample Guided Abstraction Refinement* (CEGAR), which is shown in figure 7.

The efficiency of this procedure heavily depends on the program abstraction and predicate refinement steps and several methods have been proposed to reduce the complexity. On the one side, Clarke et al. suggested an improvement to reduce the calls made to a theorem prover in the construction of the Boolean program. On the other side, Henzinger et al. [11] introduced the lazy abstraction scheme, which only refines predicates of states included in the spurious counterexample.

Interpolants are mostly used for predicate refinement after a spurious counterexample has been found. The key idea is that the reason why an abstract counter examples is a false positive in the original system is encoded in a proof of refutation [10]. In particular, Henziger et al. construct a trace formula for an abstract trace such that the formula is satisfiable if and only if the trace is feasible. In the unsatisfiable case, interpolants are constructed from the refutation (as already seen in the section on interpolants in BMC), which are then used for predicate refinement.

Subsequent to the use in predicate refinement, McMillan [16] presented a lazy interpolation based model checking procedure which directly uses interpolants for abstraction instead of predicate abstraction. Moreover, this approach generalizes the results from subsection 4.2 to infinite state systems.

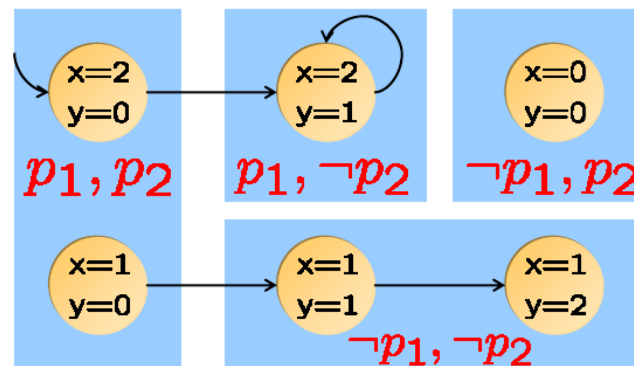



**Figure 8** A motivating example of predicate abstraction

## 5 Conclusion

In this work we have given a brief introduction on bounded model checking, as well as some of its various extensions and applications. In contrast to unbounded model checking, BMC aims towards falsification instead of verification of (safety) properties, leading to drastic performance benefits which is still meaningful for bug hunting. Furthermore, extensions for checking sequential and concurrent C programs were covered. Interpolation is an efficient methodology to enhance BMC towards the model checking of infinite state systems and verification of safety and liveness properties. Moreover, we have seen how interpolation is used for predicate abstraction and theorem proving.

## References

1 A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Nether-

lands, The Netherlands, 2009.

2 Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electr. Notes Theor. Comput. Sci.*, 66(2):160–177, 2002.

3 Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

4 Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, pages 52–71, 1981.

5 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 168–176, 2004.

6 Lucas Cordeiro and Bernd Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 331–340, New York, NY, USA, 2011. ACM.

7 William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.

8 Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification (extended abstract, category A). In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 14–26, 2003.

9 Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 72–83, 1997.

10 Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 232–244, 2004.

11 Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 58–70, 2002.

12 Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke. VCEGAR: verilog counterexample guided abstraction refinement. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 583–586, 2007.

13 Kenneth L. McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 1–13, 2003.

14 Kenneth L. McMillan. An interpolating theorem prover. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 16–30, 2004.

15 Kenneth L. McMillan. Applications of craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference,*

*TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 1–12, 2005.

16 Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 123–136, 2006.

17 Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, pages 337–351, 1982.

18 Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *In Computer-Aided Verification (CAV), LNCS 3576*, pages 82–97. Springer, 2005.

19 Klaus Schneider. Verification of reactive systems. Lecture script, Technische Universität Kaiserslautern, Gottlieb-Daimler-Str. 48, D-67663 Kaiserslautern, Germany, April 2013.

20 Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. *STTT*, 5(2-3):185–204, 2004.

21 Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 149(1):79–96, 2006.

22 Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, pages 108–125, 2000.

23 Maria Sorea. Bounded model checking for timed automata. In *Electronic Notes in Theoretical Computer Science*, page 2002. Elsevier, 2002.

24 Wikipedia. Kripke structure (model checking) — wikipedia, the free encyclopedia, 2014. [Online; accessed 29-October-2014].