Chapter 5
Explicit-State Model Checking

Gerard J. Holzmann

Abstract In this chapter we discuss the methodology used in explicit-state logic model checking, specifically as applied to asynchronous software systems. As the name indicates, in an explicit-state model checker the state descriptor for a system is maintained in explicit, and not symbolic, form, as are all state transitions. Abstraction techniques and partial-order reduction algorithms are used to reduce the search space to a minimum, and advanced storage techniques can be used to extend the reach of this form of verification to very large system sizes. The basic algorithms for explicit-state model checking date from the late 1970s and early 1980s. More advanced versions of these algorithms remain an active area of research.

5.1 Introduction

There are many different approaches that can be used for the implementation of a model-checking procedure. One of the first methods, the origins of which we can trace back to basic reachability analysis techniques that were explored as early as in the 1970s, is explicit-state model checking. Explicit-state model checking turns out to be well suited for applications in software verification, specifically the verification of systems of interacting asynchronous processes. Explicit-state techniques have gained much of their power from their integration with partial-order reduction techniques (discussed in more detail in Chap. 6 of this Handbook [25]), which make it possible to limit the number of states that must be explored to prove or disprove a property by up to an exponential factor. This has made the use of explicit-state approaches with partial-order reduction for asynchronous software systems extremely competitive when compared with the use of symbolic model-checking techniques in the verification of, for instance, synchronous circuits [2, 7, 15].

A basic reachability analysis lends itself most easily to the verification of *safety* properties, such as the validity of invariants, assertions, or the absence of deadlock in a multi-process system. As we shall see though, explicit-state model-checking

G.J. Holzmann (✉)
Nimble Research, Monrovia, CA, USA
e-mail: gholzmann@acm.org

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_5

algorithms can also be used to prove *liveness* properties, including all properties that can be formalized in Linear Temporal Logic (LTL) and more broadly the class of all ω-regular properties.

An explicit-state model-checking procedure is only possible if a few important assumptions are satisfied. First, the system that is the target of the verification must be finite state. The state of the system is a, possibly abstracted, finite tuple of values, which can be chosen from any finite domain. The system can change *state* by executing *state transitions*. This means that we assume that the system can be represented as (a set of) finite state automata. A second assumption is that system execution can be modeled as a sequence of separate state transitions. This means that even when we describe systems with multiple processes, each modeled as a finite state automaton, the effect of a system execution can be modeled by an arbitrary *interleaving* of individual process actions. With this approach we can accurately represent process scheduling on a single CPU in a multi-threaded system, where the main CPU executes one instruction at a time, interleaving the actions of different processes based on scheduling decisions. It can, however, also represent process execution on multiple CPUs, e.g., in a multi-core system, or in a networked system. Note, for instance, that also in a multi-core system it is not possible for multiple processes to access a shared memory location truly simultaneously. Although the precise ordering of read and write operations cannot always be known, at some level of granularity it is always possible to determine an interleaving order that represents the actual execution sequence. The same is not necessarily true in asynchronous hardware circuits, which therefore require a different approach to model checking.

The first attempts to build automated tools for reachability analysis targeted simple finite-state descriptions of communication protocols. In 1979, Jan Hajek used a graph exploration tool [12] to verify properties of the protocols in Tanenbaum's primer on computer networks [27]. Around the same time, Colin West and Pitro Zafiropulo developed a reachability analysis procedure for the verification of another protocol, CCITT recommendation X.21, and identified a series of flaws [29, 30]. In 1980, the first version of what later became the logic model checker Spin was developed at Bell Labs. This tool, called *pan*, was successfully used in subsequent years to expose violations of safety properties in finite-state models of telephone switching systems and communication protocols [13].

5.1.1 The Importance of Abstraction

Clearly, if we model the executions of a software system in full detail, it will generally not be feasible to perform an exhaustive verification with an explicit-state method. As a very simple example to illustrate this, consider two asynchronously executing processes each containing a single 32-bit counter. If all the two processes do is to increment their counters, executing in a simple loop, then a fully detailed model would need to represent and explore $2^{(32+32)}$ or more than 10^{19} system states. Clearly, not all those 10^{19} states are relevant to specific correctness properties that we may be interested in proving about this system. We may, for instance, want to

prove that the system does or does not terminate, or that the counter-values can wrap around the maximum. For none of these properties will it be necessary to compute all possible combinations of values held by the two counters.

The early applications to protocol verification were successful because they could be focused on the *control*-aspects of a protocol and they could abstract from the *data*-aspects. By focusing on control, one can effectively prove correctness of data transmission across unreliable channels in a way that is independent of the actual data being transmitted. An elegant example of the use of this principle is known as Wolper's data independence theorem [31]. Similarly, if we model a mutual-exclusion algorithm, the detailed computations that may be performed both outside and inside the critical section that is to be protected from multiple access are irrelevant to the correctness of the algorithm itself, and can be abstracted during the verification.

The verification process begins with the identification of the *smallest sufficient model* that allows us to prove a property of a system. What the smallest sufficient model is will generally depend on the specific property to be proven. The smallest sufficient model is generally an abstraction of the *real* system, which we refer to as the *concrete system* in what follows. There are of course requirements on the type of abstractions that can be used in this step. It should, for instance, be impossible to prove a property about the abstract system that does not hold for the corresponding concrete system, that is: the abstraction should be logically sound. In addition to abstraction, a frequently used technique in the verification of complex systems is that of restriction or specialization. A restriction reduces the abstract system to a subset of behaviors in such a way that a counter-example to a correctness claim that is generated for the restricted system is also a counter-example of the non-restricted system, but the absence of a counter-example does not logically imply the absence of a counter-example also in the non-restricted system. In practice, both abstraction (generalization) and restriction (specialization) can play a critical role in managing the complexity of software verification with explicit-state model-checking techniques. We will return to this shortly, after we discuss the basic automata-theoretic framework and the main search algorithms that are used for explicit-state model checking. More on the use of abstraction techniques in applications of logic model checking can also be found in Chap. 13 of this Handbook [9].

5.2 Basic Search Algorithms

A basic reachability analysis algorithm, sufficient for proving safety properties, is readily implemented as either a breadth-first or depth-first search. The search can be done as a basic check on all system states that are reachable from a given initial state. System states then, can be thought of as the control-flow states and variable values of a program. When we apply a model-checking algorithm to a multi-threaded system, the system state is given as a combination of local process states, and the reachability graph is the interleaving product of process actions. We start by describing this interleaving product in a little more detail.

```
 1 Open D = {}; // typically an ordered set
 2 Visited V = {};
 3
 4 start()
 5 {    V!s0; D!s0;
 6      bfs();
 7 }
 8
 9 bfs()
10 {    while (D != {})
11      {    D?s;
12           check_validity(s);
13           foreach (s, e, s') in T
14           {    if !(s' ∈ V) { V!s'; D!s'; }
15 }    }    }
```

Fig. 1 Breadth-first search

Let $A = \{S, s_0, L, T, F\}$ be a *finite state automaton*, where S is a finite set of states, s_0 is an element of S called the initial state, L is a set of symbols called the *label* set or also the *alphabet*, $T \subseteq S \times L \times S$ is a set of transitions, and $F \subseteq S$ is the set of *final* states. Automaton A is said to *accept* any finite execution that ends in any state $s \in F$.

The *interleaving product* of finite state automata $A_0, \ldots, A_N$ is another finite state automaton $A' = \{S', s'_0, L', T', F'\}$, with

- $S' = A_0.S \times A_1.S \times \cdots \times A_N.S$,
- $s'_0 = (A_0.s_0, A_1.s_0, \ldots, A_N.s_0)$,
- $L' = A_0.L \cup A_1.L \cup \cdots \cup A_N.L$,
- $T' \subseteq S' \times L' \times S'$ such that for each transition $((A_0.s_0, A_1.s_1, \ldots, A_N.s_N), e, (A_0.t_0, A_1.t_1, \ldots, A_N.t_N)) \in T'$, we have $\exists i$, $0 \leq i \leq N$, $(A_i.s_i, e, A_i.t_i) \in A_i.T$, with $e \in A_i.L$ and $\forall j \neq i$, $A_j.s_j = A_j.t_j$, and
- $F' = A_0.F \times A_1.F \times \cdots \times A_N.F$.

A basic reachability algorithm can now be constructed as shown in Fig. 1, exploring and checking all states that are reachable from the initial state s_0 of a finite automaton as defined above. The algorithm uses two data structures, a set of *open* states D and a set of *visited* states V. Open states are states that are currently being explored, with not all of their successor states visited yet. We use the following notation for operations on sets:

- $X!y$ adds y to set X; if X is ordered then it adds y as the *last* element of X,
- $X!!y$ adds y to set X; if X is ordered then it adds y as the *first* element in X,
- $X?y$ removes an element from set X and names it y; if X is ordered then the element removed is the *first* element in X, if X is empty then the operation returns the null element Ø.

The search procedure illustrated in Fig. 1 generates all states reachable from initial state s_0, while checking for violations of safety properties by calling function `check_validity()` at each state, including s_0. Set V is an unordered set. For the correctness of the basic search procedure, it does not matter if the set of open states D is ordered or not. To get a breadth-first search discipline, though, D has to

```
 1 Open D = {}; // an ordered set
 2 Visited V = {};
 3
 4 start()
 5 {    V!s0; D!s0;
 6      dfs();
 7 }
 8
 9 dfs()
10 {    if (D != {})
11      {    D?s;
12           check_validity(s);
13           foreach (s, e, s') in T
14           {    if !(s' ∈ V) { V!s'; D!!s'; dfs(); }
15 }    }    }
```

Fig. 2 Depth-first search

be ordered and used as a queue: the retrieval operation on line 11 removes the first (and oldest) element, and the add operation on line 14 adds new elements at the end.

To make it possible to generate an example execution for any state that is found to violate the property (called a *counter-example*), we can extend set V to store a pointer back to the parent state whenever a new state is added. To generate the counter-example, we then only have to follow these back-pointers to find a path from the initial state to the violating state.

At the end of the breadth-first search, set D is empty, and set V contains all states reachable in a finite number of steps from initial state s_0.

The dual of breadth-first search is depth-first search. Depth-first search is most naturally written as a recursive procedure. It is illustrated in Fig. 2. There are two key changes in this version of the algorithm compared to breadth-first search. The first change is the recursive call that is placed in the inner loop, on line 14, after a newly generated state is added to the open and visited sets, and the matching change of the surrounding while-loop into an if-statement, on line 10. The second change is the change from $D!s'$ to $D!!s'$, on line 14, which means that instead of using D as a *queue*, we now use it as a *stack*. Generating a counter-example when a safety violation is found is now simple and requires no addition to the information stored in stack D: the execution is given by the contents of stack D at the point in the search that the safety violation is detected.

Although the change looks minor, and both algorithms have the same computational complexity (both are linear in the number of reachable states) the two algorithms behave very differently. In favor of depth-first search is that the amount of information that must be stored in set D to enable counter-example generation is smaller. In favor of breadth-first search is that any counter-example generated tends to be smaller than it is for depth-first search. In fact, it is easy to show that the counter-examples generated with a breadth-first search are the shortest possible. In this case then, we have to make a tradeoff between minimizing memory use, and thus being able to handle larger problem sizes, or shortening counter-examples, and thus making it easier to understand errors found.

Note that both algorithms can work *on the fly*, which means that the reachability graph (the automaton describing the global state space as a product of the

smaller automata that formalize individual process behaviors) need not be known a priori, provided that the state transition relations are given. Both breadth-first and depth-first search can generate the product automaton on the fly, and both may be terminated as soon as a counter-example is discovered.

An important advantage of depth-first search is that it can fairly easily be extended to support not only the verification of *safety* but also *liveness* properties, without increasing the computational complexity of the search, which remains linear in the number of reachable states. Before we can discuss this extension, though, we have to discuss the extension from finite automata to ω-automata, and the fundamental relation between ω-automata and temporal logic. Both topics are covered in greater detail in other chapters of this Handbook, so a general description will suffice here.

5.3 Linear Temporal Logic

A formula in linear temporal logic (LTL) is formally defined as follows. Let f and g be arbitrary LTL formulas and let p by an arbitrary *state formula*: a Boolean expression that can be evaluated to yield a truth value for any given system state.

```
f     ::=   p | true | false | (f) | f binop f | unop f
unop  ::=   □ | ◊ | !
binop ::=   U | ∧ | ∨ | ⇒ | ⇔
```

In this grammar we have used three temporal operators: the box operator $\Box$ (which is pronounced *always*), the diamond operator $\Diamond$ (pronounced *eventually*), and the U operator *until*. The symbol $\Rightarrow$ stands for logical implication and $\Leftrightarrow$ for logical equivalence.

The semantics of temporal logic formulas is defined over infinite sequences [26]. For simplicity, we restrict ourselves here to execution sequences that start in initial state s_0.

Let σ be an infinite execution sequence, defined as the sequence of states $\sigma = s_0, s_1, \ldots, s_i, \ldots$.

$\Box$ f holds at state s_i if and only if
f holds at all states $s_j \in \sigma$ with j $\geq$ i,

$\Diamond$ f holds at state s_i if and only if
f holds for at least one state $s_j \in \sigma$ with j $\geq$ i,

f U g holds at state s_i if and only if
either g holds at s_i, or f holds at s_i and f U g holds at s_{i+1}
(informally: f is true at least until g becomes true).

5.4 Omega Automata

Any LTL formula can be mechanically converted into a Büchi automaton that accepts precisely those execution sequences for which the LTL formula is satisfied [26, 28]. The algorithm that the Spin model checker uses to convert LTL formula into Büchi automata is described in [10, 11].

The definition of a Büchi automaton $B = \{S, s_0, L, T, F\}$ is similar to the definition of a finite automaton that we presented earlier, but has a different definition of *acceptance*. Büchi automaton B accepts execution sequence σ if and only σ contains infinitely many states from set F. This means that acceptance for Büchi automata is defined for *infinite* sequences only.

A real system model may of course have both finite and infinite executions. To allow us to reason about both infinite and finite executions within the same theoretical framework we can consider any finite sequence to be a special case of an infinite sequence by extending it with an infinite repetition of its final state. This infinite repetition then corresponds to what is commonly called a *stuttering* step (a no-op) that is repeated ad infinitum, without leaving the final state. Clearly then, a finite sequence can only qualify for Büchi acceptance if the final state being stuttered is itself from set F, which matches the notion of finite acceptance from before.

Given a system A, formalized as a finite automaton, and a Büchi automaton B, formalizing all the executions of A that satisfy an LTL formula, the model-checking problem can now be phrased as the problem of finding an accepting run in the intersection of the languages accepted by the two automata. The intersection of A and B is obtained by computing the synchronous product $A \times B$, which is a Büchi automaton.

The *synchronous product* of finite state automata A and B is finite state automaton $P = \{S', s'_0, L', T', F'\}$, where $S' = A.S \times B.S$, $s'_0 = (A.s_0, B.s_0)$, $L' = A.L \times B.L$, and $T' \in S' \times L' \times S'$ such that for each transition $((A.s, B.t), (e, f), (A.s', B.t')) \subseteq T'$ we have $(A.s, e, A.s') \in A.T$, and $(B.t, f, B.t') \in B.T$.

The set of final states $F' \subseteq A.S \times B.S$ such that for each pair $(e, f) \in F'$ we have $e \in A.F \wedge f \in B.F$, i.e., each component state is a final state in its original automaton.

Since the synchronous product of two finite state automata is also finite state, an infinite execution of the product is necessarily cyclic.

Given a set of finite automata $A_0, \ldots, A_N$ and an LTL formula f, the model-checking problem can now be formalized as follows.

1. Convert LTL formula f into the corresponding Büchi automaton B.
2. Compute the interleaving product A of $A_0, \ldots, A_N$.
3. Compute the synchronous product P of $A \times B$.
4. Find accepting runs of automaton P, using the Büchi acceptance rule.

Any runs accepted by automaton P correspond to executions of A that satisfy formula f. If we are interested in finding the potential violations of f, all we have to do is to begin this procedure by replacing f with $\neg f$, using logical negation.

In practice, instead of performing steps 2, 3, and 4 one at a time, it is more efficient to perform them in a single step, again adopting an on-the-fly procedure for computing and checking P until a counter-example is found.

The model checker Spin deviates on one important point from the standard definition of the synchronous product of two Büchi automata given above. To determine the set of final states, Spin considers a state in the product $A \times B$ to be final if *at least one* of A or B is in a final state. This conforms to the standard definition in

the (standard) case where the use of final state labels (called *accept* state labels in Spin) is restricted to the property automaton B, and *all* states in the interleaving product A are considered to be final. Using the alternative definition allows Spin to use a slightly more general framework that allows also the definition of accept state labels within any of the component automata $A_0, \ldots, A_N$ (and thus A). We can then also use the model-checking procedure to find accepting runs within A itself, without requiring the definition of a separate property automaton B.

5.5 Nested Depth-First Search

The challenge is then to find infinite accepting runs in finite state automaton P. Because the accepting run must contain at least one final state, the search problem can be phrased as follows: does there exist at least one reachable final state in P that is also reachable from itself? This problem can be solved efficiently with a nested depth-first search algorithm.

The first part of this search serves to identify all final states in P that are reachable from the initial system state, and the second (nested) part of the search serves to identify those final states from this set that are also reachable from themselves. The nested search can be performed in such a way that the cost in runtime increases only by a factor of maximally two. In the worst case, each reachable state is now visited twice, but they of course only need to be stored once, which means that the memory requirements of a nested depth-first search are no different than those of a standard depth-first search. The basic procedure is illustrated in Fig. 3. The algorithm was implemented in the Spin model checker in 1989 [8], and revised to support partial order reduction in 1995 [22].

The correctness of the algorithm follows from the following theorem [8].

Theorem 1 *If acceptance cycles exist, the nested depth-first search algorithm will report at least one such cycle.*

Proof Let r be the first accepting state encountered in the depth-first search that is also reachable from itself. The nested part of the search is initiated from this state after all its successor states have been explored.

First note that state r itself cannot be reachable from any other state that was previously entered into the second state space. Suppose there was such a state w. To be in the second state space w either is itself an accepting state, or it is reachable from an accepting state. Call that accepting state v. If r is reachable from w in the second state space then r is also reachable from v. But, if r is reachable from v in the second state space, it is also reachable from v in the first state space. There are now two cases to consider:

1. The path from v to r in the first state space does not contain states that appear on the depth-first search stack when v is first reached.

```
 1 Open D = {};  // ordered set
 2 Visited V = {};
 3
 4 State seed = nil;
 5
 6 start()
 7 {    V!s0,0; D!s0,0;
 8      ndfs();    // start the first search
 9 }
10
11 ndfs()
12 {    Bit b;     // b=0: first search, b=1: nested search
13      D?s,b;
14
15      foreach (s,e,s') in T
16      {    if (s' == seed)     // seed reachable from itself
17           {     liveness_violation(); return;
18           }
19           if !(s',b∈ V) // if s' not reached before
20           {    V!s',b; D!!s',b; ndfs(); // continue search
21      }    }
22
23      // in post-order, in first search only
24
25      if (s∈ F && b == 0)
26      {    seed = s;     // a reachable final state
27           D!!s,1;       // push s on stack D
28           ndfs();       // start the nested search
29           seed = nil;   // nested search completed
30 }    }
```

Fig. 3 Nested depth-first search

2. The path from v to r in the first state space does contain at least one state x that appears on the depth-first search stack when v is first reached.

In the first case, r would have been entered into the second state space before v, due to the post-order discipline, contradicting the assumption that v is entered before r. (Remember that both r and v are assumed to be accepting states.)

In the second case, v is necessarily an accepting state that is reachable from itself, which contradicts the assumption that r is the first such state entered into the second state space.

State r is reachable from all states on the path from r back to itself, and therefore none of those states can already be in the second state space when this search begins. The path therefore cannot be truncated and r is guaranteed to find itself in the successor tree that is explored in the nested part of the search. □

It should be noted that even though the nested depth-first search algorithm visits states up to two times, each state needs to be stored just once. The states can be labeled with two bits to indicate the state space where they were first encountered, so there is virtually no increase in memory use. Another advantage of this method is that the algorithm still works *on the fly*: errors are detected during the exploration of the state space, and the search process can be cut short as soon as the first error is found.

It is clear that the computational complexity of the nested depth-first search algorithm is strictly linear in the size of P (measured as the number of reachable states

in P). The size of P itself is at most equal to the Cartesian product of the state sets of A and B. The size of A, in turn, is at most equal to the product of the sizes of all component automata $A_0, \ldots, A_N$. Finally, in the worst case the size of B can be exponential in the number of temporal operators in formula f.

In most cases of practical interest, the size of B is not a major factor: it typically contains fewer than ten states. This is in part due to the fact that most LTL formula of practical interest have very few temporal operators: the meaning of longer formulas can be notoriously hard to determine. It is also due to the fact that LTL to Büchi automata conversion algorithms very rarely exhibit worst-case behavior, although it is certainly possible to find exceptions. The real bottleneck in explicit-state logic model checking is the potential size of A.

There are two main strategies to cope with this complexity. The first is the use of partial-order reduction theory in the computation of the interleaving product A. The use of this strategy can, in the best case, achieve an exponential reduction in the size of A. If implemented well, the strategy has no down side; even in the worst case where no reduction can be achieved it will then not introduce noticeable overhead. The partial-order reduction algorithm that was implemented in the Spin model checker is described in [21, 22]. Partial-order reduction strategies are discussed in Chap. 6 of this Handbook [25]. The second main strategy is the use of abstraction, which we will discuss later in this chapter. There is also a range of coding techniques that can be used for explicit-state verification to reduce the amount of information that is stored during the verification process, thus enabling the verification of very large problems. We will also discuss some of these strategies later in this chapter.

5.6 Abstraction

To support abstraction methods (cf. Chap. 13), we can make one final change to nested depth-first search. The revised algorithm is shown in Fig. 4. The abstraction function used in this context can define a symmetry reduction [5, 6, 23], or any other abstraction, provided that it preserves logical soundness (i.e., it does not allow us to prove anything that is not true, or to disprove something that is).

Let w, x, y, and z denote states, σ and τ denote sequences of states (paths), and σ_i be the i-th state in σ. Further, let $\rightarrow$ denote the transition relation, i.e., $x \rightarrow y$ means that there exists an e such that $(x, e, y) \in T$.

A symmetric relation $\sim$ on states is a *bisimulation* relation if it satisfies the following condition [24]:

$$\forall w, y, z : (w \sim y \wedge y \rightarrow z) \Rightarrow (\exists x : w \rightarrow x \wedge x \sim z) \tag{1}$$

This means that states w and y are bisimilar if, whenever there is a transition from y to z, there is also a successor x of w such that x and z are bisimilar. We say that paths σ and τ *correspond* when $\forall i : \sigma_i \sim \tau_i$.

```
Open D = {};  // ordered set
Visited V = {};

State seed = nil;

start()
{    V!f(s0),0; D!s0,0; // f is the abstraction function
     ndfs_abstract();   // start the first search
}

ndfs_abstract()
{     Bit b; // b=0: first search, b=1: nested search

     D?s,b; // s is a concrete (not abstract) state

     foreach (s,e,s') in T
     {    if (f(s') == seed) // seed reachable from itself
          {     liveness_violation(); return;
          }
          if !(f(s'),b∈ V) // f(s') not reached before
          {     V!f(s'),b; D!!s',b; ndfs(); // continue
     }    }

     // in post-order, in first search only

     if (s ∈ F && b == 0)
     {    seed = f(s);     // a reachable final state
          D!!s,1;          // push s on stack D
          ndfs_abstract(); // start the nested search
          seed = nil;      // nested search completed
}    }
```

Fig. 4 Nested depth-first search with abstraction function $f()$

Theorem 2 ([5]) *Let $\sim$ be a bisimulation relation, and let AP be a set of state formulas such that every $P \in AP$ satisfies the condition*

$$\forall x, y : (x \sim y) \Rightarrow \big(P(x) \Leftrightarrow P(y)\big) \tag{2}$$

then any two bisimilar states satisfy the same LTL formula over the propositions in AP.

This means that any abstraction that satisfies conditions (1) and (2) preserves the logical soundness of the LTL model-checking procedure [5]. We can make use of this by defining powerful abstractions in the verification of implementation-level code to reduce what otherwise would be an overwhelming amount of computational complexity.

5.6.1 Tic-Tac-Toe

We will describe an example of the use of this type of abstraction in explicit-state model checking as it is supported by the Spin model checker. Spin is a broadly used logic model-checking tool that is based on explicit-state techniques [15]. As an example we will use the familiar game of tic-tac-toe.

```
 1 mtype = { cross, circle };
 2
 3 typedef row { mtype c[3]; }
 4 typedef square { row r[3]; }
 5 square b;
 6
 7 #define match(x,y,z)  (b.r[x].c[y] == z)
 8 #define Row(y,z)      (match(0,y,z) && match(1,y,z) && match(2,y,z))
 9 #define Column(x,z)   (match(x,0,z) && match(x,1,z) && match(x,2,z))
10 #define Up(z)         (match(2,0,z) && match(1,1,z) && match(0,2,z))
11 #define Down(z)       (match(0,0,z) && match(1,1,z) && match(2,2,z))
12 #define horizontal(z) (Row(0,z) || Row(1,z) || Row(2,z))
13 #define vertical(z)   (Column(0,z) || Column(1,z) || Column(2,z))
14 #define diagonal(z)   (Up(z) || Down(z))
15 #define try(x,y,z)    b.r[x].c[y] == 0 -> b.r[x].c[y] = z
16
17 inline check_win(z) {
18      if
19      :: horizontal(z) || vertical(z) || diagonal(z) -> end_game: 0
20      :: else // continue the game
21      fi
22 }
23
24 inline place(z) {
25      if
26      :: try(0,0,z) :: try(0,1,z) :: try(0,2,z)
27      :: try(1,0,z) :: try(1,1,z) :: try(1,2,z)
28      :: try(2,0,z) :: try(2,1,z) :: try(2,2,z)
29      // if no moves are possible, it is a draw
30      fi
31 }
32
33 init {
34      mtype symbol = cross;
35 end: do
36      :: atomic {
37           place(symbol);
38           check_win(symbol);
39           symbol = (symbol == circle -> cross : circle);
40         }
41      od
42 }
```

Fig. 5 Pure Spin model for the game of tic-tac-toe

The game itself is easily modeled. We need to model the game board as a 3×3 square board. Each place can either be empty (represented by the initial value zero), or it can contain a cross or a circle. The game proceeds by selecting an arbitrary empty place on the board, placing a symbol (cross or circle) on that place, checking for a win, and then switching sides by alternating the symbol, as shown in Fig. 5. Spin starts by using the standard C preprocessor to interpret all macro definitions and include directives. In this example we have used only macro definitions. Both macros and inlines (two of the latter are used in Fig. 5) define a purely textual expansion of function names with optional parameter replacement. The inline $place(symbol)$, for instance, is an inline call where the parameter z from the inline definition is replaced with the text $symbol$.

The initial process (named $init$) contains an infinite loop ($do \ldots od$) that contains a single indivisibly executed sequence of statements. Inline $place$ defines a nondeterministic selection ($if \ldots fi$) of the nine possible moves that each player can

make. The inline *check* finally checks whether the game is won, and if it is it will bring the execution to a halt by attempting to execute the unexecutable statement 0.

Placing a symbol on a square is modeled as a non-deterministic selection of a free space on the board, and checking for a win is simply checking for a completed row, column, or diagonal involving the last-placed symbol.

Given three possible values for any one of the nine places on the game board we have maximally 3^9 = 19,683 possible board states. Not all of these states are reachable within the game. It is, for instance, not possible to fill all the places on the board with the same symbol.

The model checker explores 5,528 states for this version of the model. We have written the model in such a way that only draw or win states are counted for the total, by grouping all actions that are part of a single move inside an atomic statement. Clearly, though, there is still a significant amount of redundancy in this set.

For every unique state of the game board there are up to eight variations of what is essentially the same state that can be obtained by rotating or mirroring the board. We can define an abstraction function f that captures this equivalence relation on board states, to reduce the amount of work that the model checker has to do to just one member of each equivalence class. If we do so, it should be possible to reduce the number of states explored to a much smaller number, and to increase the efficiency of the verification process significantly.

We can encode the state of the game board as a nine-place ternary number, by assigning place values in a fixed order, e.g., but numbering the places on the board from one to nine starting at the top left, row by row, ending at the bottom right. This final number uniquely represents the board configuration. This board state is one of 16 equivalent states though, obtained by rotation and mirroring of the board. To obtain all 16 numbers all we have to do is to assign the place values in all 16 possible ways. The abstraction function can now be defined by simply choosing one canonical representative from each set of 16 board configurations. In our example we'll use the smallest of the 16 values for the abstraction. Complexity is not a significant concern in this small example, but we will also illustrate how the abstraction can be computed in a C function that can be integrated into the model.

The Spin model checker can define transitions either as statements in the specification language, or as standard blocks of C code. No special treatment is needed to support this capability since the C code fragments merely act as *state transformers*, just like any other type of statements. The extension that supports this is very powerful though, as we will illustrate.

The extended model for the tic-tac-toe example, using C code for the abstraction function, is shown in Fig. 6. The changes from Fig. 5 are shown in **bold** (lines 5–10, and 43). First, after each move we now call the abstraction function to compute the new abstract state. This is done with a call to a C function called `abstract_value()`, which is placed in an embedded C code statement. The function returns an integer value, the abstract representation of the game board, which we assign to a new global integer variable named `abstract` in the top-level model. To be able to refer to this variable from within a C code fragment, we have to prefix this variable with the name of the state-vector (called `now.`).

```
 1 mtype = { cross, circle };
 2
 3 typedef row { mtype c[3]; }
 4 typedef square { row r[3]; }
 5 hidden square b;
 6 c_track "&b" "sizeof(struct square)" "UnMatched";
 7 int abstract;
 8 c_code {
 9       \#include "abstraction.c"
10 }
11
12 #define match(x,y,z)  (b.r[x].c[y] == z)
13 #define Row(y,z)      (match(0,y,z) && match(1,y,z) && match(2,y,z))
14 #define Column(x,z)   (match(x,0,z) && match(x,1,z) && match(x,2,z))
15 #define Up(z)         (match(2,0,z) && match(1,1,z) && match(0,2,z))
16 #define Down(z)       (match(0,0,z) && match(1,1,z) && match(2,2,z))
17 #define horizontal(z) (Row(0,z) || Row(1,z) || Row(2,z))
18 #define vertical(z)   (Column(0,z) || Column(1,z) || Column(2,z))
19 #define diagonal(z)   (Up(z) || Down(z))
20 #define try(x,y,z)    b.r[x].c[y] == 0 -> b.r[x].c[y] = z
21
22 inline check_win(z) {
23      if
24      :: horizontal(z) || vertical(z) || diagonal(z) -> end_game: 0
25      :: else // continue the game
26      fi
27 }
28
29 inline place(z) {
30      if
31      :: try(0,0,z) :: try(0,1,z) :: try(0,2,z)
32      :: try(1,0,z) :: try(1,1,z) :: try(1,2,z)
33      :: try(2,0,z) :: try(2,1,z) :: try(2,2,z)
34      // if no moves are possible, it?s a draw
35      fi
36 }
37
38 init {
39      mtype symbol = cross;
40 end: do
41      :: atomic {
42           place(symbol);
43           c_code { now.abstract = abstract_value(); };
44           check_win(symbol);
45           symbol = (symbol == circle -> cross : circle);
46         }
47      od
48 }
```

Fig. 6 Spin model for the game of tic-tac-toe using an abstraction function

The only other issue that we now have to address is that with only the change above the model checker would see two representations of the same state: the concrete representation from before and the abstract representation that we just added. Clearly we will need both, since the concrete state determines at each point which moves are valid, and we need the abstract state to determine whether we are exploring a previously unseen board state. The method we can use in Spin to accomplish this exploits the fact that depth-first search uses two different data structures: the set of open states D (also known as the search stack) and the set of visited states V (also known as the state space). The nested depth-first search with abstraction illustrated in Fig. 4 stores the abstract states in V and the concrete states in D. The only thing

we need to be able to do, then, is to effect that the concrete state of the game board, `square b` in Fig. 5, is not stored in V but in D.

Hiding the concrete state of `square b` from both the state vector and the stack is done with declaration primitive `hidden`. But this accomplishes too much, since we do want to track the state of `square b` in search stack D. We can do this with the help of a `c_track` statement.

The first argument to `c_track` is a pointer to the data structure that we want to track. Because `square b` is declared `hidden`, this is a simple reference to `b`, without the need for a prefix (i.e., it is not part of the state vector). The second argument specifies the size of the object being tracked, which in this case is simply the size of data structure `square`. For the third argument we use the keyword `UnMatched`, which indicates that indeed the value of this object is not part of the system state as stored in the state space. The only alternative option for the third argument would be the keyword `Matched`, which however would completely undo the effect of declaration prefix `hidden`. In this case it seems redundant to have to use both `hidden` in the data declaration and `UnMatched` in the `c_track` statement, but note that in general `c_track` statements can be used to track the state of any data object, including those that are declared in C code external to the model checker where we often need the capability to consider this external data to be part of the state as stored in state space V.

We can now complete the model by including the abstraction function itself, which is written in C here, into the model. We can do this with an embedded `c_code` statement, which in Fig. 6 is placed immediately following the declaration of the new variable abstract.

The revised model using the abstraction function reaches 765 states, down from the original 5,528 states, matching the exact number of uniquely different board positions [1]. The abstraction we have used is logically sound (satisfying conditions (1) and (2) above), which means that the abstract model can prove precisely the same properties as the non-abstract version.

5.7 Model-Driven Verification

In the process of constructing the abstract version of the tic-tac-toe model, we have also seen how an explicit-state model checker can be used to track data that is declared outside the model itself, in C code, and how it can call an external C function to perform a state transition (e.g., one that modifies the externally declared tracked data). This is a powerful concept that makes it possible to use abstraction in full LTL verification of implementation level code.

In this *model-driven verification* approach, the non-determinism that is supported by the model checker can be used to drive an application into all relevant states, as reflected in the values of the tracked variables. An abstraction function is again used to compute an abstract representation of the data being tracked and stores it in canonical form. A linked list data structure, for example, consists of both the data

that is stored in the list elements and the pointers that connect the list elements. An abstraction function can collect just the list data elements into a simple array, preserving their order while abstracting from all pointer values. Note that the pointer values have no significance other than to fix the order of the list elements. Two linked lists with the same contents stored in two different places in memory would differ as concrete objects (because the pointer values differ), but they can trivially be recognized as equivalent under this abstraction. Arbitrary LTL properties (both safety and liveness) can be proven in this way with explicit-state model-checking techniques for even large applications, while using abstraction. Any counter-example generated is immediately also a concrete counter-example, because the concrete values for the full error trace are available on the search stack.

5.8 Incomplete Storage

One of the benefits of the explicit-state model-checking procedure is that we can store a different representation of system states in state space V than we do on search stack D. When abstraction is used, the state space can be restricted to storing only abstract values, which in general take up less space than the concrete values they represent. But we can also use other storage techniques to exploit this capability. We can, for instance, use lossless compression techniques. An attractive aspect here is that the compression need not be reversible.

Given that we have considerable freedom in choosing a storage strategy, it is natural to ask at this point what would happen if we permitted the state compression technique to be lossy. If $f(s)$ represents the compression function for state s, then a lossless compression function has the property that

$$\forall s, s',\ f(s) = f\left(s'\right) \Rightarrow s = s'. \tag{3}$$

We lose this property if $f()$ is allowed to be lossy.

5.8.1 Bitstate Hashing and Bloom Filters

Consider the case where we use f to compute an N-bit hash function of the bit-representation of the state-descriptor. No matter what the size of s is, $f(s)$ then always returns an N-bit representation of it. To store the state now requires setting just 1 bit in a 2^N-bit memory arena. For $N = 32$, for instance, 2^{32} bits or 512 Mbyte suffices to store all reachable states explicitly after this type of possibly lossy compression. There are several reasons why this can be attractive. One reason is that it requires only a fixed amount of memory, independent of the actual problem size. A second reason is that the storage operation itself is very fast, given that it takes only one single-bit operation. But what are the possible effects of any hash collisions?

A hash collision occurs when two different states ($s \neq s'$) produce the same compressed value ($f(s) = f(s')$). If this happens, the model checker will conclude (Fig. 4) that the new state was previously visited, when in fact it was not. The search is truncated and some reachable system states can remain unvisited. The search becomes incomplete. Despite this incompleteness, one very important property of the model-checking procedure is preserved though: when a counter-example is found it will always be an accurate counter-example to the property being checked. Counter-examples can be missed, but those that are found are uncorrupted. In considering this limitation it is important to realize that also a search that performs lossless storage (with or without abstraction) can become incomplete. A computer always has only finite memory and when all available memory has been used the search will come to a stop. If the fraction of all reachable states that can be explored with the hash method is larger than the fraction that can be explored with a lossless storage technique, then the lossy technique is clearly preferred. A small example can illustrate this.

Consider a verification model with a reachable state space of one million states of 1024 bytes each. Storing all states explicitly would require 1 Gbyte of memory. If we have only 128 Mbyte of memory available then we can explore no more than approximately 10% of all states in a single run. We can use the same 128 Mbyte of available memory, though, as a hash arena and use a hash function that compresses each state to 27 bits which can be used directly as bit addresses in the 128-Mbyte arena. This hash arena would have room to store up to 10^9 states, of which we will use just 0.1%.

With a good quality hash function we therefore statistically have a good probability to explore *all* of the 10^6 reachable states, giving us full problem coverage in most cases. Even though this makes this search method (implemented in most explicit-state model checkers as a *bitstate* or *supertrace* search mode) attractive, we have to trade certainty of problem coverage for a statistical expectation.

The search method we have described was introduced in 1987 as the *bitstate hashing* or *supertrace* method. Over the years, it has proven to be a very effective tool in handling large verification models in applications of explicit-state model checking [14]. The theoretical justification for the method turns out to be very similar to that of a storage method that was first described by Burton Bloom in 1970 [4]. The algorithm, or variations of it, e.g., the hash-compact method [32], have been implemented in almost all explicit-state model-checking systems.

5.9 Extensions

The basic explicit-state model-checking procedure we have described so far can be extended in many different ways to support different types of optimizations. The specific type of optimization to be chosen will generally depend on which critical resource use needs to be reduced. Two relevant extensions to explicit-state model checking that have been studied recently are, for instance,

- Context-bounded model checking: To search for those counter-examples that contain the fewest number of context switches. The enforcement of this constraint in a model-checking procedure generally increases run-time and memory use [19].
- Multi-core model-checking algorithms: The objective of these search algorithms is to reduce the run-time requirements of a model-checking run by leveraging the computational power of larger numbers of processing cores (or CPUs in a grid or cloud network) [3, 16–18, 20].

5.10 Synopsis

In this chapter we have presented an overview of explicit-state logic model-checking procedures, specifically those based on an automata-theoretic framework. We have illustrated how this approach supports the use of powerful abstraction techniques and, through the use of embedded C code to specify parts of a logic model, how it can be combined with methods for the direct verification of implementation level artifacts.

References

1. The game of tic-tac-toe. http://f2.org/maths/ttt.html
2. Avrunin, G., Corbett, J., Dwyer, M., Pasareanu, C., Siegel, S.: Benchmarking finite-state verifiers. Int. J. Softw. Tools Technol. Transf. **2**(4), 317–320 (2000)
3. Barnet, J., Brim, L., Rockai, P.: DiVinE multi-core, a parallel LTL model checker. In: Liu, Z., Ravn, A.P. (eds.) Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 5799, pp. 234–239. Springer, Heidelberg (2009)
4. Bloom, B.: Spacetime trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
5. Bosnacki, D.: Enhancing state space reduction techniques for model checking. Ph.D. thesis, Eindhoven University of Technology (2001)
6. Clarke, E., Emerson, E., Jha, S., Sistla, A.: Symmetry reduction in model checking. In: Hu, A.J., Vardi, M.Y. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1427, pp. 147–158. Springer, Heidelberg (1998)
7. Corbett, J.: Evaluating deadlock detection methods for concurrent software. Trans. Softw. Eng. **22**(3), 161–180 (1996)
8. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. Form. Methods Syst. Des. **1**(2–3), 275–288 (1992)
9. Dams, D., Grumberg, O.: Abstraction and abstraction refinement. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
10. Etessami, K., Holzmann, G.: Optimizing Büchi automata. In: Palamidessi, C. (ed.) Proc. 11th Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1877, pp. 153–167. Springer, Heidelberg (2000)
11. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proc. of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, pp. 3–18. Chapman & Hall, London (1996)
12. Hajek, J.: Automatically verified data transfer protocols. In: Intl. Conf. on Computer Communication (ICCC), pp. 749–756 (1978)

13. Holzmann, G.: PAN: a protocol specification analyzer. Tech. Rep. TM81-11271-5, AT&T Bell Laboratories, (1981)
14. Holzmann, G.: An improved reachability analysis technique. Softw. Pract. Exp. **18**(2), 137–161 (1988)
15. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
16. Holzmann, G.: Parallelizing the Spin model checker. In: Donaldson, A.F., Parker, D. (eds.) Intl. Workshop on Model Checking Software (SPIN). LNCS, vol. 7385, pp. 155–171. Springer, Heidelberg (2012)
17. Holzmann, G.: Proving properties of concurrent programs. In: Bartocci, E., Ramakrishnan, C.R. (eds.) Intl. Symposium on Model Checking of Software (SPIN). LNCS, vol. 7976, pp. 18–23. Springer, Heidelberg (2013)
18. Holzmann, G., Bosnacki, D.: The design of a multi-core extension of the Spin model checker. Trans. Softw. Eng. **33**(10), 659–674 (2007)
19. Holzmann, G., Florian, M.: Model checking with bounded context switching. Form. Asp. Comput. **23**(3), 365–389 (2011)
20. Holzmann, G., Joshi, R., Gorce, A.: Swarm verification techniques. Trans. Softw. Eng. **37**(6), 845–857 (2011)
21. Holzmann, G., Peled, D.: An improvement in formal verification. In: Proc. of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII, pp. 197–211. Chapman & Hall, London (1995)
22. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: Grégoire, J.C., Holzmann, G., Peled, D. (eds.) The Spin Verification System. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 23–32. DIMACS/AMS, Providence (1996)
23. Ip, C., Dill, D.: Better verification through symmetry. Form. Methods Syst. Des. **9**(1–2), 41–75 (2006)
24. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-Conference on Theoretical Computer Science. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
25. Peled, D.: Partial-order reduction. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
26. Pnueli, A.: The temporal logic of programs. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 46–57. IEEE, Piscataway (1977)
27. Tanenbaum, A.: Computer Networks, 1st edn. Prentice Hall, New York (1981)
28. Vardi, M., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994). Journal version of a conference paper first published in 1983
29. West, C.: General technique for communications protocol validation. IBM J. Res. Dev. **22**(3), 393–404 (1978)
30. West, C., Zafiropulo, P.: Automated validation of a communications protocol: the CCITT X.21 recommendation. IBM J. Res. Dev. **22**(1), 60–71 (1978)
31. Wolper, P.: Specifying interesting properties of programs in propositional temporal logic. In: Symp. on Principles of Programming Languages (POPL), pp. 184–193. ACM, New York (1986)
32. Wolper, P., Leroy, D.: Reliable hashing without collision detection. In: Courcoubetis, C. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 697, pp. 59–70. Springer, Heidelberg (1993)