Model Checking and the State Explosion Problem

Edmund M. Clarke[1], William Klieber[1], Miloš Nováček[2], and Paolo Zuliani[1]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
[2] ETH Zürich, Zürich, Switzerland

Abstract. Model checking is an automatic verification technique for concurrent systems that are finite state or have finite state abstractions. It has been used successfully to verify computer hardware, and it is beginning to be used to verify computer software as well. As the number of state variables in the system increases, the size of the system state space grows exponentially. This is called the "state explosion problem". Much of the research in model checking over the past 30 years has involved developing techniques for dealing with this problem. In these lecture notes, we will explain how the basic model checking algorithms work and describe some recent approaches to the state explosion problem, with an emphasis on Bounded Model Checking.

1 Introduction

Ensuring the correctness of software and hardware is an issue of great importance, as failures often cause considerable financial losses and can even have fatal consequences in safety-critical systems. This has led to an increased interest in applying formal methods and verification techniques in order to develop high-assurance systems. Among the most successful techniques that are widely used in both research and industry is *model checking*.

Model checking is a collection of automatic techniques for verifying finite-state concurrent systems. This framework was developed independently in the early 1980's by Clarke and Emerson [7] and by Queille and Sifakis [21]. Traditionally, model checking has been mainly applied to hardware. However, thanks to the tremendous progress that the research community has made over the past three decades, model checking has been successfully applied to many aspects of software verification. The main obstacle that model checking faces is the *state explosion problem*. The number of global states of a concurrent system with multiple processes can be enormous; it is exponential in both the number of processes and the number of components per process.

* This research was sponsored by the GSRC under contract no. 1041377 (Princeton University), the National Science Foundation under contracts no. CNS0926181 and no. CNS0931985, the Semiconductor Research Corporation under contract no. 2005TJ1366, General Motors under contract no. GMCMUCRLNV301, the Air Force Office of Scientific Research (BAA 2011-01), and the Office of Naval Research under award no. N000141010188.

In these lecture notes, we will explain how the basic model checking algorithms work and describe some recent approaches to the state explosion problem, focusing in particular on *bounded model checking*.

1.1 Organization of these Lecture Notes

The lecture notes are organized as follows. In Section 2 we describe how *reactive systems* can be modeled as *finite-state transition systems*. Section 3 provides an overview of *temporal logic*, the formal notation in which we specify the desired behavior of a system. We concentrate on three main temporal logics: Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and CTL*. In Section 4 we define the model checking problem and provide an overview of a basic model-checking algorithm for CTL. We also summarize the main advantages and disadvantages of model checking. In Section 5 we introduce the main disadvantage of model checking — *the state explosion problem.* We will explain what this problem is, what its implications are, and what key breakthroughs have been made over the past three decades. Finally, in Section 6 we focus on one such breakthrough: Bounded Model Checking (BMC). In particular, BMC has overcome a number of issues from which model checking suffers, and it has enabled the application of model checking techniques to a wide range of software and hardware systems. Section 6 describes BMC and its application in greater detail.

2 Modeling Concurrent Systems

In order to reason about the correctness of a system, one needs to specify the properties that the system should satisfy, *i.e.*, the intended behavior of the system. Once we have such a specification we need to construct a *formal model* for the system. Such a model should capture the basic properties that must be considered in order to verify the behavioral specification (the so called *atomic propositions*). Furthermore, to make the verification simpler, the model should abstract away all the details that have no effect on the correctness with respect to the specification.

In these notes we shall focus on concurrent (*reactive*) systems and their temporal behavior. A reactive system may frequently need to interact with its environment (including the user), and often do not terminate. A simple, functional model is thus not adequate to model the behavior of a reactive system. Instead, we use *Kripke structures*, a type of state-transition graphs.

A Kripke structure consists of a finite set of states, a set of state transitions (*i.e.*, a relation over the states), and a labeling function which assigns to each state the set of atomic propositions that are true in this state. In such a model, at any point in time the system is in one of the possible states, and the transition relation describes how the system moves from a state to another over one time step. The formal definition is the following.

Definition 1. *A **Kripke structure** (or **state transition system**) M is a quadruple $M = \langle S, S_0, R, L \rangle$ consisting of*

1. *a finite set of states S;*
2. *a set of initial states $S_0 \subseteq S$;*
3. *a total transition relation $R \subseteq S \times S$; that is, for every state $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$;*
4. *a set AP of atomic propositions (Boolean functions over S) and a labeling function $L : S \to 2^{AP}$ that labels each state with the set of atomic propositions that hold in that state.*

A *path* π in M from a state s_0 is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ where for all $i \geq 0$, $(s_i, s_{i+1}) \in R$.

3 Temporal Logic

Given a state transition system (Kripke structure) M, we would like to reason about certain properties that the system should satisfy. For example, we might want to ask the following questions:

- If we start from any initial state of M, is it possible to reach an error state[1]?
- Is every request eventually acknowledged?
- Can the system be always restarted[2]?

In this section we describe several formalisms for specifying temporal properties of reactive systems. In particular, we focus on *temporal logics*, which can be used to concisely describe properties of paths generated by Kripke structures. For example, a temporal logic formula might specify that some particular state is *eventually* reached in a path, or that an error state is *never* reached in any path. Such properties are specified using *temporal operators*. Here we will focus on the following temporal logics:

- *Linear Temporal Logic* (LTL), in which temporal operators are provided for describing events along a single computation path (linear time).
- *Computation Tree Logic* (CTL), in which the temporal operators quantify over the paths that are possible from a given state (branching time).
- CTL*, an extension of CTL that combines both branching-time and linear-time operators.

We first describe CTL*, as it is the most expressive of the three temporal logics mentioned above.

3.1 CTL*

CTL* [8] subsumes both LTL and CTL. Formulas in CTL* describe properties of *computation trees*. The tree represents all the possible paths of a Kripke structure, and it is formed by infinitely unwinding the state transition graph from the initial state as illustrated in Fig. 1.

[1] This is an example of a *safety property*.
[2] This is an instance of a *liveness property*.

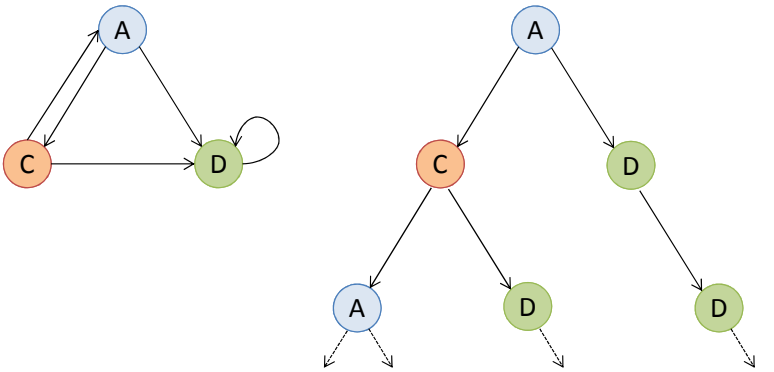

Fig. 1: Computation Tree. The initial state of the state transition system on the left is the state labeled A.

CTL* formulas consist of atomic propositions, Boolean connectives, *path quantifiers*, and *temporal operators*. There are two path quantifiers, **A** and **E**, that describe the branching structure of the computation tree:

- **A**f ("for all computational paths") is true iff the formula f holds along all the paths in the computation tree;
- **E**f ("for some computation path") is true iff the formula f is true along some path in the computation tree.

The path quantifiers **A** and **E** are dual, in fact:

$$\mathbf{A}f \equiv \neg\mathbf{E}(\neg f).$$

There are five basic temporal operators:

- **X** f ("next time") is true iff the formula f holds in the second state of the path.
- **F** f ("eventually", "in the future" or "sometimes") is true iff the formula f will hold at some state on the path.
- **G** f ("always" or "globally") is true iff the formula f holds at every state on the path.
- f **U** g ("until") is true iff there is a state on the path where g holds and at every preceding state on the path, f holds.

There are two types of CTL* formulas: *state formulas*, whose truth is defined over *states*, and *path formulas*, whose truth is defined over *paths*. We recall that CTL* formulas are interpreted over a given Kripke structure (Definition 1).

The syntax of *state* formulas is defined as follows:

- an atomic proposition $p \in AP$ is a state formula,

- if f and g are state formulas, then $\neg f$, $f \vee g$, and $f \wedge g$ are state formulas,
- if f is a path formula, then $\mathbf{A}f$ and $\mathbf{E}f$ are state formulas.

The syntax of *path* formulas is defined as follows:

- if f is a state formula, then f is also a path formula,
- if f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}\, f$, $\mathbf{F}\, f$, $\mathbf{G}\, f$, and $f\ \mathbf{U}\ g$ are path formulas.

We have the following formal definition. It is customary to assume that AP contains *true*, the atomic proposition which identically holds.

Definition 2 (CTL*). *The syntax of CTL* is given by the grammar:*

$$\begin{array}{ll} \phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{A}\psi \mid \mathbf{E}\psi & \text{(state } \textit{formulas}\text{)} \\ \psi ::= \phi \mid \neg\psi \mid \psi \vee \psi \mid \mathbf{X}\,\psi \mid \mathbf{F}\,\psi \mid \mathbf{G}\,\psi \mid \psi\ \mathbf{U}\ \psi & \text{(path } \textit{formulas}\text{)} \end{array}$$

where $p \in AP$.

We now define the semantics of CTL*. Let π^i denote the *suffix* of path π starting from the i-th state. Given a state transition M and a state formula ϕ, the notation $M, s_0 \models \phi$ is interpreted as "in M at state s_0, the formula ϕ is true". Similarly, given a path formula ψ, the notation $M, \pi \models \psi$ is interpreted as "in M, the formula ψ is true along the path π". Let ϕ_1 and ϕ_2 be state formulas and let ψ_1 and ψ_2 be path formulas. The semantics of CTL* is as follows:

$$\begin{array}{lcl}
M, s \models p & \Leftrightarrow & p \in L(s) \\
M, s \models \neg\phi_1 & \Leftrightarrow & M, s \not\models \phi_1 \\
M, s \models \phi_1 \vee \phi_2 & \Leftrightarrow & M, s \models \phi_1 \text{ or } M, s \models \phi_2 \\
M, s \models \phi_1 \wedge \phi_2 & \Leftrightarrow & M, s \models \phi_1 \text{ and } M, s \models \phi_2 \\
M, s \models \mathbf{E}\ \psi_1 & \Leftrightarrow & \text{there exists a path } \pi \text{ from } s \text{ such that } M, \pi \models \psi_1 \\
M, s \models \mathbf{A}\ \psi_1 & \Leftrightarrow & \text{for all paths } \pi \text{ from } s,\ M, \pi \models \psi_1 \\
M, \pi \models \phi_1 & \Leftrightarrow & M, s \models \phi_1, \text{ where } s \text{ is the first state of } \pi \\
M, \pi \models \neg\psi_1 & \Leftrightarrow & M, \pi \not\models \psi_1 \\
M, \pi \models \psi_1 \vee \psi_2 & \Leftrightarrow & M, \pi \models \psi_1 \text{ or } M, \pi \models \psi_2 \\
M, \pi \models \psi_1 \wedge \psi_2 & \Leftrightarrow & M, \pi \models \psi_1 \text{ and } M, \pi \models \psi_2 \\
M, \pi \models \mathbf{X}\,\psi_1 & \Leftrightarrow & M, \pi^1 \models \psi_1 \\
M, \pi \models \mathbf{F}\,\psi_1 & \Leftrightarrow & \text{there exists } k \geq 0 \text{ such that } M, \pi^k \models \psi_1 \\
M, \pi \models \mathbf{G}\,\psi_1 & \Leftrightarrow & \text{for all } i \geq 0,\ M, \pi^i \models \psi_1 \\
M, \pi \models \psi_1\ \mathbf{U}\ \psi_2 & \Leftrightarrow & \text{there exists } k \geq 0 \text{ such that } M, \pi^k \models \psi_2 \\
 & & \text{and for all } 0 \leq j < k,\ M, \pi^j \models \psi_1 \\
 & & \text{exists a } k \leq j \text{ such } \pi^k \models \psi_2
\end{array}$$

Note that we can define the temporal operators $\mathbf{F}$ and $\mathbf{G}$ using only $\mathbf{U}$ in the following way (syntactic sugar):

$$\begin{array}{rcl} \mathbf{F}\, f & \equiv & \textit{true}\ \mathbf{U}\ f \\ \mathbf{G}\, f & \equiv & \neg\mathbf{F}\,(\neg f) \end{array}$$

Also, one can show that any CTL* formula can be written by using the operators $\neg, \vee, \mathbf{X}, \mathbf{U}$, and $\mathbf{E}$.

3.2 Linear Temporal Logic (LTL)

Pnueli was the first to use temporal logic for reasoning about concurrent programs [20]. In particular, Pnueli used *linear temporal logic* (LTL), a sublogic of CTL*. Formulas in LTL are of the form $\mathbf{A}f$, where f is a LTL path formula as follows:

- if $p \in AP$, then p is a path formula;
- if f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}\, f$, $\mathbf{F}\, f$, $\mathbf{G}\, f$, and $f\ \mathbf{U}\ g$ are path formulas.

Therefore, LTL restricts state formulas to be just atomic propositions.

Definition 3 (LTL). *Linear temporal logic formulas are of the form* $\mathbf{A}\psi$*, with* ψ *given by the grammar:*

$$\psi ::= p \mid \neg\psi \mid \psi \vee \psi \mid \mathbf{X}\,\psi \mid \psi\ \mathbf{U}\ \psi$$

where $p \in AP$*.*

The semantics of LTL formulas is the same as for CTL* path formulas.

Example 1 (LTL formulas). Note that since LTL formulas are always quantified by an outer $\mathbf{A}$, the LTL formula $\mathbf{F}\,\mathbf{G}\,p$ is the same as $\mathbf{A}(\mathbf{F}\,\mathbf{G}\,p)$ in CTL*. We can write $\mathbf{G}\,(req \Rightarrow \mathbf{F}\,ack)$ for a property specifying that every *request* in execution is eventually *acknowledged*. The formula $(\mathbf{G}\,\mathbf{F}\,enabled) \Rightarrow (\mathbf{G}\,\mathbf{F}\,executed)$ specifies that if an event is infinitely often *enabled* then it will be infinitely often *executed*. (The 'implication' operator $\Rightarrow$ is defined by $\psi_1 \Rightarrow \psi_2 \equiv \neg\psi_1 \vee \psi_2$.)

3.3 Computation Tree Logic (CTL)

Another useful sublogic of CTL* is CTL [7, 21]. In CTL, each basic temporal operator must be immediately preceded by a path quantifier (*i.e.*, either $\mathbf{A}$ or $\mathbf{E}$). In particular, CTL can be obtained from CTL* by restricting the form of path formulas as follows:

- if f and g are *state* formulas, then $\mathbf{X}\,f$, $\mathbf{F}\,f$, $\mathbf{G}\,f$, and $f\ \mathbf{U}\ g$, are path formulas.

Definition 4 (CTL). *Computation tree logic formulas are inductively defined as follows:*

$$\begin{aligned} \phi &::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{A}\psi \mid \mathbf{E}\psi \quad \text{(state } \textit{formulas}\text{)} \\ \psi &::= \mathbf{X}\,\phi \mid \mathbf{F}\,\phi \mid \mathbf{G}\,\phi \mid \phi\ \mathbf{U}\ \phi \quad \text{(path } \textit{formulas}\text{)} \end{aligned}$$

where $p \in AP$*.*

Example 2 (CTL formulas). We can write $\mathbf{EF}\,(WhiteWins)$ for a property specifying that there is a way for a white player to win, and $\mathbf{AG}\,(\mathbf{EF}\,Restart)$ can be written for a property specifying that it is always possible to restart from any state.

It can be shown that any CTL formula can be written in terms of $\neg, \vee, \mathbf{EX}\,, \mathbf{EG}\,,$ and $\mathbf{EU}$. The semantics of four widely used CTL operators is exemplified in Fig. 2.

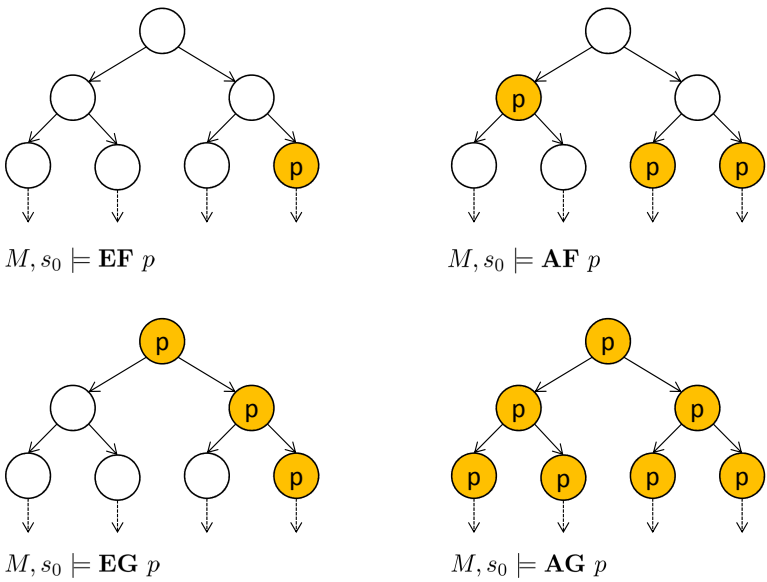

Fig. 2: Basic CTL operators.

Note that in CTL the equivalence $\mathbf{A}f \equiv \neg\mathbf{E}\neg f$ does not hold in general. The equivalence does hold for $\mathbf{X}\,, \mathbf{G}\,,$ and $\mathbf{F}\,$. However, for $\mathbf{U}$ we have that:

$$\mathbf{A}[f\ \mathbf{U}\ g] \equiv \neg\mathbf{E}[\neg g\ \mathbf{U}\ (\neg f \wedge \neg g)] \wedge \neg\mathbf{EG}\,\neg g\ .$$

3.4 Expressiveness of Temporal Logics

Even though it might seem that CTL is more expressive than LTL, it is not the case. For example, the LTL formula $\mathbf{A}(\mathbf{F}\,\mathbf{G}\,p)$ cannot be expressed in CTL. Similarly, the CTL formula $\mathbf{AG}\,(\mathbf{EF}\,p)$ cannot be expressed in LTL. From these examples it follows that CTL and LTL are incomparable. However, they are both sublogics of CTL*, and in fact CTL* is strictly more expressive than both LTL and CTL. For example, the CTL* formula $\mathbf{A}(\mathbf{F}\,\mathbf{G}\,p) \vee \mathbf{AG}\,(\mathbf{EF}\,q)$ cannot be expressed in either CTL or LTL. The expressiveness of these three logics is illustrated in Fig. 3.

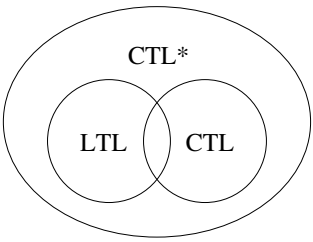

Fig. 3: Expressiveness of CTL, LTL, and CTL*.

4 Model Checking

Since its development in the early 1980's, model checking has been applied to a large number of problems, such as complex sequential circuit designs and communication protocols. Model checking overcomes a number of problems that other approaches based on simulation, testing, and deductive reasoning suffer from. To mention a few, approaches based on testing are not complete, and deductive reasoning using theorem provers is generally not fully automated since it has much higher complexity. On the other hand, *model checkers* are 'push-button' software tools, they do not require any proofs, and they can provide *diagnostic counterexamples* when a universally path-quantified specification (i.e., a specification of the form $\mathbf{A}\phi$ where ϕ is a CTL* formula) is found to be false. Thanks to these and other features, model checkers have become very popular for (hardware) verification, and are also often used for debugging purposes.

A model checker (see Fig. 4) is usually composed of three main parts:

1. a *property specification language* based on a temporal logic;
2. a *model specification language* — a formal notation for encoding the system to be verified as a finite-state transition system, *i.e.*, the *model*;
3. a *verification procedure* — an intelligent exhaustive search of the model state space that determines whether the specification is satisfied or not. In the latter case, the procedure provides a counterexample path exhibiting the violation of the specification.

We can define the model checking problem as follows.

Definition 5. *Let M be a state-transition graph and let f be a temporal logic formula. The model checking problem is to find all the states $s \in S$ such that $M, s \models f$.*

In [7], Clarke and Emerson introduced CTL and presented a corresponding verification procedure. In an independent work, Queille and Sifakis [21] introduced similar ideas. We now briefly explain a CTL verification procedure more efficient than the original one presented in [7].

Recall that we are given a Kripke structure $M = \langle S, S_0, R, L\rangle$ and a CTL formula f. Our task is to compute the set $\{s \in S \mid M, s \models f\}$. The CTL

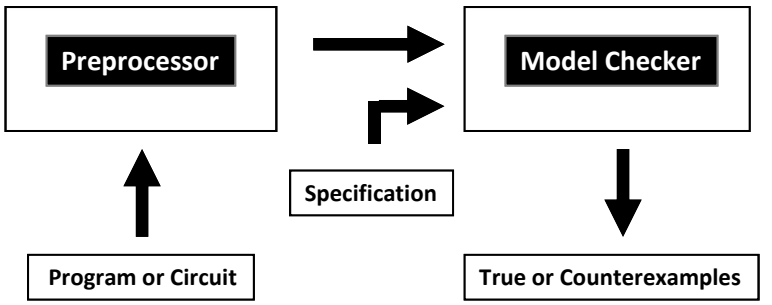

Fig. 4: A Model Checker with Counterexamples

model checking algorithm works by computing for each state s the set $label(s)$ of subformulas of f that are true in s. When the computation of such sets is finished, we will have that $M, s \models f$ iff $f \in labels(s)$. To compute each set $label(s)$ we proceed as follows. Initially, every state is labeled with the atomic propositions that hold in the state, *i.e.*, $label(s) = L(s)$, for all $s \in S$. We then proceed recursively to label each state with the subformulas of f that hold in that state, starting from the most deeply nested subformulas, and moving outward to finish with f itself. Since any CTL formula is expressible in terms of $\neg, \vee, \mathbf{EX}, \mathbf{EU}$, and $\mathbf{EG}$, the verification procedure need only to consider these types of CTL formulas.

The Boolean operators $\neg$ and $\vee$ are easily handled. For a formula of the type $\neg f$, we label (by "$\neg f$") the states that are not labeled by f. For a disjunction $f \vee g$, we label (by "$f \vee g$") the states that are labeled by f or by g. To handle formulas of the form $\mathbf{EX}\, f$, we label the states from which there exists a transition to a state labeled by f.

For $\mathbf{E}[f\ \mathbf{U}\ g]$, we first need to find all the states labeled by g. Then, from those states we follow the transition relation backwards (*i.e.*, using R^{-1}) to find all the states that can be reached by a path in which every state is labeled by f. The states selected in this way are labeled by "$\mathbf{E}[f\ \mathbf{U}\ g]$". In Fig.4 we give pseudocode for a procedure CheckEU that implements the labeling for formulas of the type $\mathbf{E}[f\ \mathbf{U}\ g]$. Of course, it assumes that the subformulas f and g have been already processed, *i.e.*, all the states satisfying either f or g have been labeled accordingly.

The procedure for $\mathbf{EG}\, f$ is based on the decomposition of the state-transition graph into strongly connected components, and it is slightly more complex. The interested reader can find details in [11, Chapter 4].

It can be shown that the time complexity of the algorithm is $O(|f|\cdot(|S|+|R|))$, where $|f|$ is the number of different subformulas of f. Because the algorithm explicitly accesses all the states of the transition system, this approach is also known as *explicit-state* model checking.

```
function CheckEU(f, g) {
  T := {s | g ∈ label(s)};
  for all s ∈ T
      label(s) := label(s) ∪ {E[f U g]};
  while T ≠ ∅ {
      choose s ∈ T;
      T := T \ {s};
      for all t such that R(t,s) {
          if E[f U g] ∉ label(t) and f ∈ label(t) {
              label(t) := label(t) ∪ {E[f U g]};
              T := T ∪ {t};
          }
      }
  }
}
```

Fig. 5: Explicit-state procedure for labeling the states satisfying $\mathbf{E}[f\ \mathbf{U}\ g]$.

4.1 Pros and Cons

Model checking is a very powerful framework for verifying specifications of finite-state systems. One of the main advantages of model checking is that it is fully automated. No expert is required in order to check whether a given finite-state model conforms to a given set of system specifications. Model checking also works with partial specifications, which are often troublesome for techniques based on theorem proving. When a property specification does not hold, a model checker can provide a counterexample (an initial state and a set of transitions) that reflects an actual execution leading to an error state. This is the reason why tools based on model checking are very popular for debugging.

One aspect that can be viewed as negative is that model checkers do not provide correctness proofs. Another negative aspect is that model-checking techniques can be directly applied only to finite-state systems. An infinite-state system can by abstracted into a finite model; however, this leads to a loss of precision. Perhaps the most important issue in model checking is the *state-explosion problem*. It is apparent from the complexity of the CTL model checking algorithm that its practical usefulness critically depends on the size of the state space. Basically, if number of states grows too large, so does the complexity of the verification procedure, possibly making the technique unusable. In the next Section we focus on the state explosion problem and on several possible methods to combat it.

5 State Explosion Problem

The number of states of a model can be enormous. For example, consider a system composed by n processes, each having m states. Then, the asynchronous composition of these processes may have m^n states. Similarly, in a n-bit counter,

the number of states of the counter is exponential in the number of bits, *i.e.*, 2^n. In model checking we refer to this problem as the *state explosion* problem. All model checkers suffer from it. Using arguments from complexity theory, it can be shown that, in the worst case, this problem is inevitable. However, researchers have developed many techniques that address the state explosion problem. These techniques are frequently used in industrial applications of model checking. In this section, we will concentrate on key advances that make model checking a practical technique in both research and industry.

There have been several major advances in addressing the state explosion problem. One of the first major advances was *symbolic model checking* with *binary decision diagrams* (BDDs). In this approach, a set of states is represented by a BDD instead of by listing each state individually. The BDD representation is often exponentially smaller in practice. Model checking with BDDs is performed using a *fixed point* algorithm. Another major advance is the *partial order reduction*, which exploits independence of actions in a system with asynchronous composition of processes. A third major advance is *counterexample-guided abstraction refinement*, which adaptively tries to find an appropriate level refinement, precise enough to verify the property of interest yet not burdened with irrelevant detail that slows down verification. Finally, *bounded model checking* exploits fast Boolean satisfiability (SAT) solvers to search for counterexamples of bounded length. In this Section we give a brief overview of the first three techniques, while bounded model checking is explained in greater detail in the next Section.

5.1 Fixed-Point Algorithms

The symbolic model-checking algorithm is based on fixpoint characterizations of the basic temporal operators. For simplicity, we will consider only CTL model checking, although similar ideas can used for LTL model checking (see, e.g., Section 6.7 of [11]). Let $M = \langle S, S_0, R, L\rangle$ be a finite state-transition system. The set $\mathcal{P}(S)$ of all subsets of S forms a lattice under the set inclusion ordering. For convenience, we identify a state formula with the set of states in which it is true. For example, we identify the formula *false* with the empty set of states, and we identify the formula *true* with S (the set of all states). Each element of $\mathcal{P}(S)$ can be viewed both as a set of states and as a state formula (a *predicate*). A function $\tau : \mathcal{P}(S) \to \mathcal{P}(S)$ will be called a *predicate transformer*.

Definition 6. We say that a state formula f is the *least fixed point* (or respectively *greatest fixed point*) of a predicate transformer τ iff (1) $f = \tau[f]$, and (2) for all state formulas g, if $g = \tau[g]$, then $f \subseteq g$ (respectively $f \supseteq g$).

Definition 7. A predicate transformer τ is *monotonic* iff for all $f, g \in \mathcal{P}(S)$ $f \subseteq g$ implies $\tau(f) \subseteq \tau(g)$.

A monotonic predicate transformer on $\mathcal{P}(S)$ always has a least fixed point and a greatest fixed point (by Tarski's Fixed Point Theorem [23]). The temporal operators **AF** , **EF** , **AU**, and **EU** can each be characterized as the least fixed point

```
function Lfp(τ) {
    Q := false;
    while (Q ≠ τ(Q)) {
        Q := τ(Q);
    }
    return Q;
}
```

```
function Gfp(τ) {
    Q := true;
    while (Q ≠ τ(Q)) {
        Q := τ(Q);
    }
    return Q;
}
```

Fig. 6: Procedures for computing least and greatest fixed points

of a monotonic transformer. Similarly, the temporal operators **AG** and **EG** can each be characterized as the greatest fixed point of a monotonic transformer:

- **AF** f is the least fixed point of $\tau[Z] = f \vee \mathbf{AX}\, Z$.
- **EF** f is the least fixed point of $\tau[Z] = f \vee \mathbf{EX}\, Z$.
- **AG** f is the greatest fixed point of $\tau[Z] = f \wedge \mathbf{AX}\, Z$.
- **EG** f is the greatest fixed point of $\tau[Z] = f \wedge \mathbf{EX}\, Z$.
- $\mathbf{A}[f\ \mathbf{U}\ g]$ is the least fixed point of $\tau[Z] = g \vee (f \wedge \mathbf{AX}\, Z)$.
- $\mathbf{E}[f\ \mathbf{U}\ g]$ is the least fixed point of $\tau[Z] = g \vee (f \wedge \mathbf{EX}\, Z)$.

We can calculate the least fixed point of τ as follows. We define $U_0 = \emptyset$ and $U_i = \tau(U_{i-1})$ for $i \geq 1$. We first compute U_1, then U_2, then U_3, and so on, until we find a k such that $U_k = U_{k-1}$. It can be proved that the U_k computed in this manner is the least fixed point of τ. To compute the greatest fixed point, we follow a similar procedure. Pseudocode for both procedures is shown in Fig. 6. In Fig. 7 we illustrate the computation for **EF** p.

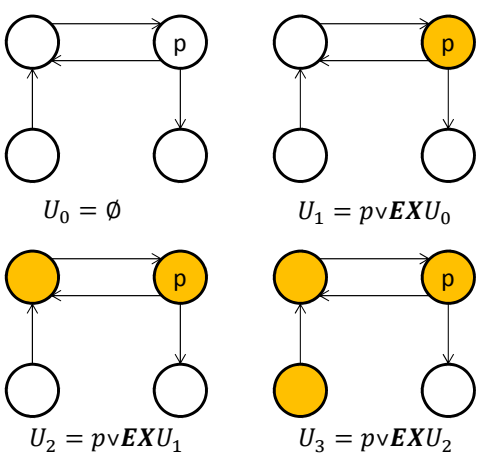

Fig. 7: Example of computing fixed point

5.2 Symbolic Model Checking with OBDDs

The main idea behind symbolic model checking is to represent and manipulate a finite state-transition system symbolically as a Boolean function. In particular, *Ordered binary decision diagrams* (OBDDs) [3] are a canonical form for Boolean formulas. OBDDs are often substantially more compact than traditional normal forms. Moreover, they can be manipulated very efficiently.

We consider Boolean formulas over n variables $x_1, \ldots, x_n$. A *binary decision diagram* (BDD) is a rooted directed acyclic graph with two types of vertexes, *terminal vertices's* and *nonterminal vertexes*. Each nonterminal vertex v is labeled by a variable $var(v)$ and has two successors, $low(v)$ and $high(v)$. Each terminal vertex v is labeled by either 0 or 1 via a Boolean function $value(v)$. A BDD with root v determines a Boolean function $f_v(x_1, \ldots, x_n)$ in the following manner:

- If v is a terminal vertex then $f_v(x_1, \ldots, x_n) = value(v)$.
- If v is a nonterminal vertex with $var(v) = x_i$ then $f_v(x_1, \ldots, x_n)$ is given by

$$\Big(\neg x_i \wedge f_{low(v)}(x_1, \ldots, x_n)\Big) \ \vee \ \Big(x_i \wedge f_{high(v)}(x_1, \ldots, x_n)\Big).$$

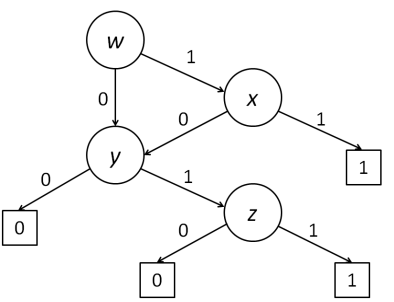

Fig. 8: OBDD for the formula $(w \wedge x) \vee (y \wedge z)$, with ordering $w < x < y < z$.

In an OBDD there is a strict total ordering of the variables $x_1, \ldots, x_n$ when traversing the diagram from the root to the terminals. In Fig. 8 we illustrate the OBDD for the formula $(w \wedge x) \vee (y \wedge z)$, with variable ordering $w < x < y < z$. Given an assignment to the variables w, x, y, and z, the value of the formula can be decided by traversing the OBDD from the root to the terminals. At each node, branching is decided by the value assigned to the variable that labels the node. For example, the assignment $(w = 0, x = 1, y = 0, z = 1)$ generates in the OBDD the traversal $w \xrightarrow{0} y \xrightarrow{0} 0$, so the formula does not hold for this assignment.

In practical applications, it is desirable to have a *canonical representation* for Boolean functions. This simplifies tasks like checking equivalence of two formulas

and deciding if a given formula is satisfiable or not. Such a representation must guarantee that two Boolean functions are logically equivalent if and only if they have isomorphic representations. Two binary decision diagrams are *isomorphic* if there exists a bijection $\mathcal{H}$ between the graphs such that

- terminals are mapped to terminals, and nonterminals to nonterminals,
- for every terminal vertex v, $value(v) = value(\mathcal{H}(v))$, and
- for every nonterminal vertex v:
  - $var(v) = var(\mathcal{H}(v))$,
  - $\mathcal{H}(low(v)) = low(\mathcal{H}(v))$, and
  - $\mathcal{H}(high(v)) = high(\mathcal{H}(v))$.

A canonical representation for Boolean functions can be obtained by placing two restrictions on binary decision diagrams [3]:

- The variables appear in the same order along each path from the root to a terminal. (We write $x < y$ to denote that x is prior to y in this ordering.)
- There are no isomorphic subtrees or redundant vertexes in the diagram.

There exist efficient algorithms for operating on OBDDs. We begin with the function that restricts some argument x_i of the Boolean function f to a constant value b. This function is denoted by $f|_{x_i \leftarrow b}$ and satisfies the identity

$$f|_{x_i \leftarrow b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).$$

If f is represented as an OBDD, the OBDD for the restriction $f|_{x_i \leftarrow b}$ is computed by a depth-first traversal of the OBDD. For any vertex v which has an edge with a vertex w such that $var(w) = x_i$, we replace the edge by $low(w)$ if b is 0 and by $high(w)$ if b is 1. If doing so renders vertex v redundant (i.e., if $high(w)$ becomes equal to $low(w)$), we must remove the redundancy to preserve canonicity. We must also take care not to create a new vertex that is isomorphic to an existing vertex.

All sixteen two-argument logical operations can be implemented efficiently on Boolean functions that are represented as OBDDs. In particular, the complexity of these operations is linear in the product of the size of the argument OBDDs. The key idea for efficient implementation of these operations is the *Shannon expansion*

$$f = (\neg x \wedge f|_{x \leftarrow 0}) \ \vee \ (x \wedge f|_{x \leftarrow 1}).$$

Bryant [3] gives a uniform procedure called *Apply* for computing all 16 logical operations. Let $\star$ be an arbitrary two-argument logical operation, and let f and f' be two Boolean functions. To simplify the explanation of the algorithm we introduce the following notation:

- v and v' are the roots of the OBDDs for f and f';
- $x = var(v)$ and $x' = var(v')$.

We consider several cases depending on the relationship between v and v'.

- If v and v' are both terminal vertexes, then $f \star f' = value(v) \star value(v')$.

- If $x = x'$, then we use the Shannon expansion

$$f \star f' \;=\; \big(\neg x \wedge (f|_{x\leftarrow 0} \;\star\; f'|_{x\leftarrow 0})\big) \vee \big(x \wedge (f|_{x\leftarrow 1} \;\star\; f'|_{x\leftarrow 1})\big)$$

  to break the problem into two subproblems, which are solved recursively. The root of the resulting OBDD will be a new node r with $var(r) = x$, while $low(r)$ will be the OBDD for $(f|_{x\leftarrow 0} \star f'|_{x\leftarrow 0})$ and $high(r)$ will be the OBDD for $(f|_{x\leftarrow 1} \star f'|_{x\leftarrow 1})$.
- If $x < x'$, then $f'|_{x\leftarrow 0} = f'|_{x\leftarrow 1} = f'$ since f' does not depend on x. We thus have that

$$f \star f' \;=\; \big(\neg x \wedge (f|_{x\leftarrow 0} \;\star\; f')\big) \vee \big(x \wedge (f|_{x\leftarrow 1} \;\star\; f')\big)$$

  and the OBDD for $f \star f'$ is computed recursively as in the second case.
- If $x' < x$, then the required computation is similar to the previous case.

The algorithm is made polynomial by using memoization:

- a hash table is used to record all previously computed subproblems;
- before any recursive call, the table is checked to see if the subproblem has been solved. If it has, the result is obtained from the table; otherwise, the recursive call is performed.
- The result must be reduced to ensure that it is in canonical form.

In Fig.9 we illustrate the construction of the canonical OBDD form for a simple formula. It should be noted that the size of BDDs depends greatly on the chosen variable order. For example, the BDD representing an n-bit comparator is linear with a good ordering, but exponential with a bad ordering.

Symbolic Model Checking with BDDs. Ken McMillan implemented a version of the CTL model checking algorithm using BDDs in the fall of 1987. Subsequently, much larger concurrent systems could be handled than with explicit-state model checking [5, 4]. State-transition graphs can be represented with BDDs as follows. First, we must represent the states in terms of n Boolean state variables $\boldsymbol{v} = \langle v_1, v_2, \ldots, v_n\rangle$. Then, we express the transition relation R as a Boolean formula in terms of the state variables:

$$f_R(v_1, \ldots, v_n, v'_1, \ldots, v'_n) = 1 \quad \text{iff} \quad R(v_1, \ldots, v_n,\, v'_1, \ldots, v'_n)$$

where $v_1, \ldots v_n$ represents the current state and $v'_1, \ldots, v'_n$ represents the next state. Finally, we convert f_R to a BDD.

We define a procedure *ToBDD* that takes a CTL formula $f(\boldsymbol{v})$ and returns a BDD that represents exactly those states of the system that satisfy the CTL formula. We define *ToBDD* inductively over the structure of CTL formulas. If $f(\boldsymbol{v})$ is an atomic proposition, then $ToBDD(f(\boldsymbol{v}))$ is the BDD representing the set of states satisfying the atomic proposition. If $f(\boldsymbol{v})$ has the form $g_1 \star g_2$ for some Boolean operator $\star$, then $ToBDD(f(\boldsymbol{v})) = Apply(ToBDD(g_1) \star ToBDD(g_2))$. Temporal operators are handled as follows:

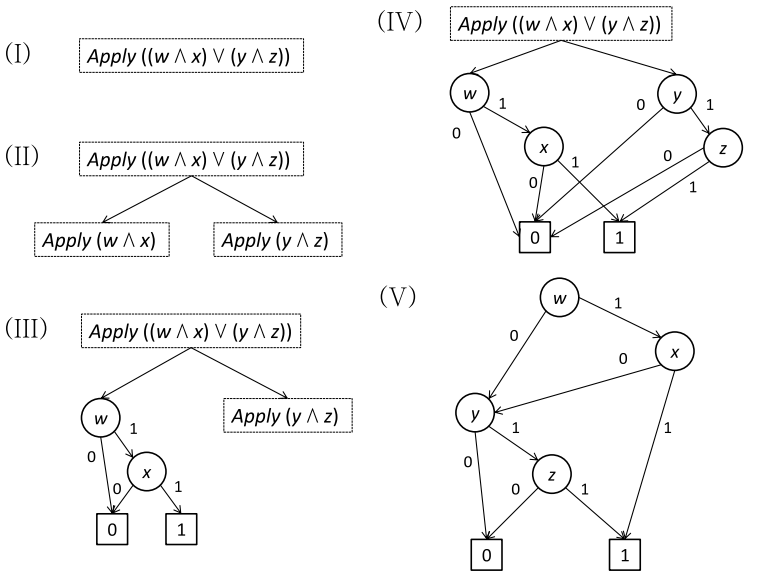

Fig. 9: Step-by-step construction of the canonical OBDD for the formula $(w \wedge x) \vee (y \wedge z)$, using ordering $w < x < y < z$.

- $\mathit{ToBDD}(\mathbf{EX}\, f(\boldsymbol{v})) = \mathit{ToBDD}(\exists \boldsymbol{v}'.\ f_R(\boldsymbol{v}, \boldsymbol{v}') \wedge f(\boldsymbol{v}'))$
- $\mathit{ToBDD}(\mathbf{EF}\, f(\boldsymbol{v})) = \mathit{Lfp}(\lambda Z.\ \mathit{ToBDD}(f(\boldsymbol{v}) \vee \mathbf{EX}\, Z))$
- $\mathit{ToBDD}(\mathbf{E}[f(\boldsymbol{v})\ \mathbf{U}\ g(\boldsymbol{v})]) = \mathit{Lfp}(\lambda Z.\ \mathit{ToBDD}(g(\boldsymbol{v}) \vee (f(\boldsymbol{v}) \wedge \mathbf{EX}\, Z)))$

If $f(\boldsymbol{v})$ has the form $\exists v_i.\, g$, then $\mathit{ToBDD}(f(\boldsymbol{v})) = \mathit{ToBDD}(g|_{v_i \leftarrow 0} \vee g|_{v_i \leftarrow 1})$. Finally, to check whether a formula $f(\boldsymbol{v})$ holds true in a set of initial states $I(\boldsymbol{v})$, we check whether the formula $I(\boldsymbol{v}) \Rightarrow f(\boldsymbol{v})$ holds.

5.3 Partial Order Reduction

As we have already mentioned, asynchronous composition of processes in a concurrent system may cause exponential blow-up of the system state space. This is an even bigger problem in software verification than in hardware verification. The reason is that software tends to be less structured than hardware. Hence, the state explosion problem has been the main obstacle in applying model checking to software.

One of the most successful techniques for dealing with asynchronous systems is the *partial order reduction*. This technique is based on the observation that many events are independent of each other, and can be executed in arbitrary order without affecting the outcome of the computation. This means that it is possible to avoid exploring certain paths in the state-transition system. In Fig. 10 we show an example of two independent paths, $(s \rightarrow s_0 \rightarrow s')$ and $(s \rightarrow s_1 \rightarrow s')$,

assuming that the variables `x` and `y` are not aliases. Therefore, it is enough to explore only one of the two paths.

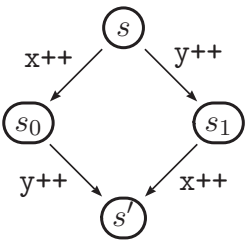

Fig. 10: Partial order reduction and independent events.

One of the big challenges for the partial order reduction is that the reduction must be on the fly and we must *locally* decide which transitions can be safely ignored, as it is not feasible to construct the whole transition system first and then prune it.

5.4 Counterexample-Guided Abstraction Refinement (CEGAR)

If the model state space is very large, or even infinite, performing an exhaustive search of the entire space is not feasible. Therefore, when building the model we should try to abstract only the relevant information from the (concrete) state-transition system. The *counterexample-guided abstraction refinement* (CEGAR) [9] technique uses counterexamples to refine an initial abstraction.

Let $M = \langle S, s_0, R, L\rangle$ be a Kripke structure. We write $M_\alpha = \langle S_\alpha, s_0^\alpha, R_\alpha, L_\alpha\rangle$ to denote the *abstraction* of M with respect to an *abstraction mapping* α. We assume that the states of both M and M_α are labeled with atomic propositions from a set AP. A function $\alpha : S \to S_\alpha$ is an *abstraction mapping* from M to M_α with respect to a set of atomic propositions $A_\alpha \subseteq AP$ iff the following three conditions hold true:

1. $\alpha(s_0) = s_0^\alpha$
2. If there is a transition from s to t in M, then there is a transition from $\alpha(s)$ to $\alpha(t)$ in M_α.
3. For all states s, $L(s) \cap A_\alpha = L_\alpha(\alpha(s))$.

In Fig. 11 we illustrate a concrete system and its abstraction. The key theorem relating abstract and concrete systems was proved by Clarke, Grumberg, and Long [10]: an abstraction preserves all the true formulas of a certain fragment of CTL*.

Theorem 1 (Property Preservation Theorem). *If a universal CTL* property holds on the abstract model, then it holds also on the concrete model.*

A universal CTL* formula must not contain existential path quantifiers (**E**) when written in negation normal form. For example, **AG** f is a universal formula, while **EG** f is not.

It is easy to show that the converse of the Property Preservation Theorem is not true, *i.e.*, there are universal properties which holds in the concrete system but fail in the abstract system. Therefore, a counterexample to the property in the abstract system may not be a counterexample in the concrete system. Such counterexamples are called *spurious*. Given a counterexample in the abstract system, we can decide whether it is spurious or not simply by executing it on the concrete system. If the counterexample checks on the concrete system, *i.e.*, it is not spurious, then we have found an actual violation of the property and thus a bug in the system. If the counterexample is spurious, then we use it to refine the abstraction mapping, and we check again the property on the more precise abstraction. We continue this process until there are no spurious counterexamples.

In general, the presence of spurious counterexamples cannot be avoided, since the abstract model over-approximates the state space of the concrete system. This is due to the loss of information caused by the abstraction mapping. However, the state space of the abstract system is usually much smaller than that of the concrete system, making the abstract system amenable to model checking.

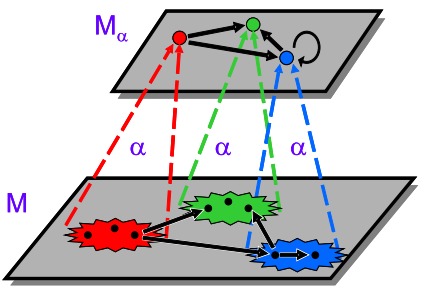

Fig. 11: A concrete system M and its abstraction M_α.

6 Bounded Model Checking

Bounded Model Checking (BMC) [1] is the method used by most industrial-strength model checkers today. Given a finite state-transition system, a temporal logic property, and a bound k, BMC generates a propositional formula that is satisfiable if and only if the property can be disproved by a counterexample of length k. This propositional formula is then fed to a *Boolean satisfiability* (SAT) solver. If no counterexample of length k is found, then we look for longer counterexamples by incrementing the bound k. For safety properties (*i.e.*, checking whether a "bad" state is unreachable), it can be shown that we only need to check counterexamples whose length is smaller than the *diameter* of the system — the smallest number of transitions to reach all reachable states. Alternatively,

BMC can be used for bug catching (rather than full verification) by simply running it up to a given counterexample length or for a given amount of time. BMC has several advantages over symbolic model checking with BDDs:

1. BMC finds counterexamples faster than BDD-based approaches.
2. BMC finds counterexamples of minimal length.
3. BMC uses much less memory than BDD-based approaches.
4. BMC does not require the user to select a variable ordering and does not need to perform costly dynamic reordering.

In BMC, the states of the model are represented as vectors of Booleans. For example, in a hardware circuit, the state of each flip-flop would be usually encoded as a single Boolean variable. A state transition system is encoded as follows:

- the set of initial states is specified by a propositional formula $I(s)$ that holds true iff s is an initial state;
- the transition relation is specified by a propositional formula $R(s, s')$ that holds true iff there exists a transition from s to s';
- for each atomic proposition p, there is a propositional formula $p(s)$ that holds true iff p is true in state s.

Definition 8. *A sequence of states $(s_0, ..., s_k)$ is a **valid path prefix** iff:*

1. *$I(s_0)$ holds true (s_0 is an initial state); and*
2. *$\bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$ holds true (for all $i < k$ there exists a transition $s_i \rightarrow s_{i+1}$)*

For simplicity, we first describe BMC for LTL safety properties of the form $\mathbf{G}\, p$, where p is an atomic proposition.

6.1 Safety Properties

The property $\mathbf{G}\, p$ asserts that p holds true in all reachable states (remember that LTL formulas are implicitly quantified by an outer $\mathbf{A}$ path operator.) We wish to determine whether there exists a counterexample whose length is no larger than a fixed bound k. In other words, we wish to determine whether there exists a valid path prefix $(s_0, ..., s_k)$ in which p fails for some state s_i, with $i \leq k$. Thus, we have that a sequence $(s_0, ..., s_k)$ is a counterexample to $\mathbf{G}\, p$ iff the following formula is satisfiable:

$$\underbrace{I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})}_{(s_0,\ldots,s_k) \text{ valid path prefix}} \wedge \underbrace{\bigvee_{i=0}^{k} \neg p(s_i)}_{p \text{ fails in } (s_0,\ldots,s_k)} \tag{1}$$

Example 3. We write $s[i]$ to denote bit i of the state $s = \langle s[0] \ldots s[n] \rangle$. Consider a 3-bit state transition system defined by

$$\begin{aligned} I(s) &= \neg s[0] \wedge \neg s[1] \wedge \neg s[2] \\ R(s, s') &= (s[2] \Leftrightarrow s'[1]) \wedge (s[1] \Leftrightarrow s'[0]) \\ p(s) &= \neg s[0] \ . \end{aligned}$$

We want to model check the property $\mathbf{G}\, p$. First we try to find a counterexample of length $k = 0$. (We measure length of a path prefix by the number of transitions between states, not the number of states; a counterexample of length 0 is a sequence of exactly one state.) Substituting into formula (1), we obtain:

$$I(s_0) \wedge \neg p(s_0) \;=\; (\neg s_0[0] \wedge \neg s_0[1] \wedge \neg s_0[2]) \wedge s_0[0]$$

which is clearly unsatisfiable, so no counterexample of length 0 exists. It turns out that the shortest counterexample is of length 3. In fact, for $k = 3$ we have that formula (1) becomes

$$\begin{aligned}(\neg s_0[0] \wedge \neg s_0[1] \wedge \neg s_0[2]) \wedge (s_0[2] \Leftrightarrow s_1[1]) \wedge (s_0[1] \Leftrightarrow s_1[0]) \\ \wedge (s_1[2] \Leftrightarrow s_2[1]) \wedge (s_1[1] \Leftrightarrow s_2[0]) \\ \wedge (s_2[2] \Leftrightarrow s_3[1]) \wedge (s_2[1] \Leftrightarrow s_3[0]) \\ \wedge (s_0[0] \vee s_1[0] \vee s_2[0] \vee s_3[0])\end{aligned}$$

which is satisfiable by the states $(s_0, s_1, s_2, s_3) = (\langle 000\rangle, \langle 001\rangle, \langle 011\rangle, \langle 111\rangle)$. Therefore, the sequence of state transitions $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$ is a counterexample to $\mathbf{G}\, p$.

In practice, the formulas obtained by expanding (1) can be very large. Nevertheless, BMC remains useful because modern SAT solvers can efficiently handle formulas with millions of clauses.

6.2 Determining the Bound

We now discuss two methods for determining the counterexample length when verifying a safety property such as $\mathbf{G}\, p$. Let d be the *diameter* of the system, *i.e.*, the least number of steps to reach all reachable states. Alternatively, d is the least number for which the following holds: for every state s, if there exists a valid path prefix that contains s (*i.e.*, s is reachable), then there exists a valid path prefix of length at most d that contains s. Clearly, if property p holds for all valid path prefixes of length k, where $k \geq d$, then p holds for all reachable states. So, we only need to consider counterexamples of length at most d. However, finding d is computationally hard.

Given a bound k, we can decide whether $k \geq d$ by solving a *quantified Boolean formula*. In particular, if every state reachable in $k+1$ steps can also be reached in up to k steps, then $k \geq d$. More formally, let $\mathit{reach}_{=n}$ and $\mathit{reach}_{\leq n}$ be the predicates defined over the state space S as follows:

$$\mathit{reach}_{=n}(s) \;=\; \exists s_0, ..., s_n \quad I(s_0) \wedge \bigwedge_{i=0}^{n-1} R(s_i, s_{i+1}) \wedge s = s_n$$

$$\mathit{reach}_{\leq n}(s) \;=\; \exists s_0, ..., s_n \quad I(s_0) \wedge \bigwedge_{i=0}^{n-1} R(s_i, s_{i+1}) \wedge \left(\bigvee_{i=0}^{n} s = s_i \right)$$

The predicate $reach_{=n}(s)$ holds iff s is reachable in *exactly* n transitions, while $reach_{\leq n}$ holds iff s can be reached in *no more* than n transitions. Then, $k \geq d$ iff

$$\forall s \in S \quad reach_{=k+1}(s) \Rightarrow reach_{\leq k}(s) \ . \tag{2}$$

The above method of bounding the counterexample length is of limited value due to the difficulty of solving the quantified Boolean formula (2). Another way of using BMC to prove properties (*i.e.*, not merely for bug-finding) is *k-induction* [22]. With k-induction, to prove a property $\mathbf{G}\, p$, one needs to a find an *invariant* q such that:

1. $q(s) \Rightarrow p(s)$, for all $s \in S$.
2. For every valid path prefix $(s_0, ..., s_k)$, $q(s_0) \wedge ... \wedge q(s_k)$ holds true.
3. For every state sequence $(s_0, ..., s_{k+1})$, if $\bigwedge_{i=0}^{k} R(s_i, s_{i+1})$ holds true then $(q(s_0) \wedge ... \wedge q(s_k)) \Rightarrow q(s_{k+1})$ holds true.

Other techniques for making BMC complete are cube enlargement [17], circuit co-factoring [13], and Craig interpolants [18].

6.3 BMC for General LTL Properties: Original Encoding

In this Section we present the BMC encoding for full LTL, as originally proposed by Biere *et al.* [1]. A counterexample to $\mathbf{F}\, p$ can only be an infinite path. In order to use a finite path prefix to represent an infinite path, we consider potential *back-loops* from the last state of a finite path prefix to an earlier state, as illustrated in Fig. 12. More precisely, a valid path prefix $(s_0, ..., s_\ell, ..., s_k)$ has a *back-loop* from k to ℓ iff the transition relation R contains the pair (s_k, s_ℓ).

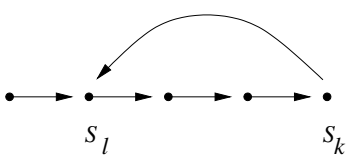

Fig. 12: A lasso-shaped path

Note that an LTL formula is false iff its negation is true. So, the problem of finding a counterexample of an LTL formula f is equivalent to the problem of finding a witness to its negation $\neg f$. In this section, we will follow this approach.

Given a state transition system M, an LTL formula f, and a bound k, we will construct a propositional formula $[\![M, f]\!]_k$ that holds true iff there exists a path prefix $(s_0, ..., s_k)$ along which f holds true. We assume that all negations in f have been pushed inward so that they occur only directly in front of atomic propositions. First we define a propositional formula $[\![M]\!]_k$ that constrains $(s_0, ..., s_k)$ to be a valid path prefix:

$$[\![M]\!]_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \ . \tag{3}$$

Now, we have to consider two cases, depending on whether the sequence $(s_0, ..., s_k)$ has a back-loop or not. First we consider the case without a back-loop. We introduce a bounded semantics, employing the following identities (similar to those used in the fixed-point characterizations of CTL discussed in Section 5.1):

- $\mathbf{F}\, f \ = \ f \vee \mathbf{X}\, \mathbf{F}\, f$
- $\mathbf{G}\, f \ = \ f \wedge \mathbf{X}\, \mathbf{G}\, f$
- $[f\ \mathbf{U}\ g] \ = \ g \vee (f \wedge \mathbf{X}\, [f\ \mathbf{U}\ g])$

Definition 9 (Bounded Semantics without a Back-Loop). Given a bound k and a finite or infinite sequence π whose first k states are $(s_0, ..., s_k)$, we say that an LTL formula f *holds true along* π *with bound* k iff $\pi \models_k^0$ is true, where $\pi \models_k^i$ is defined recursively as follows for $i \in \{0, ..., k\}$:

$\pi \models_k^i \ p$	iff	atomic proposition p is true in state s_i
$\pi \models_k^i \ \neg p$	iff	atomic proposition p is false in state s_i
$\pi \models_k^i \ f \vee g$	iff	$(\pi \models_k^i f)$ or $(\pi \models_k^i g)$
$\pi \models_k^i \ f \wedge g$	iff	$(\pi \models_k^i f)$ and $(\pi \models_k^i g)$
$\pi \models_k^i \ \mathbf{X}\, f$	iff	$i < k$ and $(\pi \models_k^{i+1} f)$
$\pi \models_k^i \ \mathbf{F}\, f$	iff	$\pi \models_k^i f \vee \mathbf{X}\, \mathbf{F}\, f$
$\pi \models_k^i \ \mathbf{G}\, f$	iff	$\pi \models_k^i f \wedge \mathbf{X}\, \mathbf{G}\, f$
$\pi \models_k^i \ f\ \mathbf{U}\ g$	iff	$\pi \models_k^i g \vee \big(f \wedge \mathbf{X}\, [f\ \mathbf{U}\ g]\big)$

Note that the recursion is well-founded, since $\pi \models_k^i \mathbf{X}\, f$ is false if $i \geq k$. This also means that formulas of the type $\mathbf{G}\, f$ do not hold true for any bound.

It is easily seen that $\pi \models_k^0 f$ implies $\pi \models f$ for any infinite path π and LTL formula f. Given a bound k, an LTL formula f, and a valid path prefix $(s_0, ..., s_k)$, we construct a propositional formula $[\![f]\!]_k^0$ that is true iff $\pi \models_k^0 f$.

Definition 10 (Original translation of LTL formula without a loop).

$$
\begin{aligned}
[\![p]\!]^i_k &:= p(s_i) \quad \text{where } p \text{ is an atomic proposition} \\
[\![\neg p]\!]^i_k &:= \neg p(s_i) \quad \text{where } p \text{ is an atomic proposition} \\
[\![f \vee g]\!]^i_k &:= [\![f]\!]^i_k \vee [\![g]\!]^i_k \\
[\![f \wedge g]\!]^i_k &:= [\![f]\!]^i_k \wedge [\![g]\!]^i_k \\
[\![\mathbf{X}\, f]\!]^i_k &:= \begin{cases} [\![f]\!]^{i+1}_k & \text{if } i < k \\ \mathsf{false} & \text{otherwise} \end{cases} \\
[\![\mathbf{F}\, f]\!]^i_k &:= [\![f]\!]^i_k \vee [\![\mathbf{X}\, \mathbf{F}\, f]\!]^i_k \\
[\![\mathbf{G}\, f]\!]^i_k &:= [\![f]\!]^i_k \wedge [\![\mathbf{X}\, \mathbf{G}\, f]\!]^i_k \\
[\![f\ \mathbf{U}\ g]\!]^i_k &:= [\![g]\!]^i_k \vee \left([\![f]\!]^i_k \wedge [\![\mathbf{X}\, (f\ \mathbf{U}\ g)]\!]^i_k \right)
\end{aligned}
$$

The translations for $\mathbf{F}$ and $\mathbf{G}$ are easily expanded:

$$
\begin{aligned}
[\![\mathbf{F}\, f]\!]^i_k &= \bigvee_{j=i}^{k} [\![f]\!]^j_k \\
[\![\mathbf{G}\, f]\!]^i_k &= \mathsf{false}\ .
\end{aligned}
$$

For $[\![f\ \mathbf{U}\ g]\!]^i_k$, we write a propositional formula that requires that g holds for some path suffix π^j (where $i \leq j \leq k$) and that f holds on all path suffixes in the set $\{\pi^n \mid i \leq n < j\}$, as illustrated in Fig. 13:

$$
[\![f\ \mathbf{U}\ g]\!]^i_k = \bigvee_{j=i}^{k} \left([\![g]\!]^j_k \wedge \bigwedge_{n=i}^{j-1} [\![f]\!]^n_k \right) .
$$

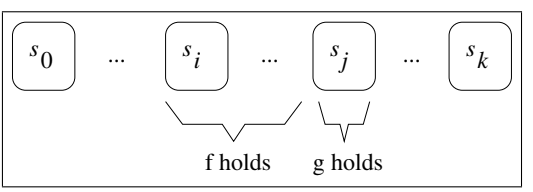

Fig. 13: Translation of $[\![f\ \mathbf{U}\ g]\!]^i_k$ for a loop-free path prefix.

Now consider a path prefix $(s_0, ..., s_k)$ with a back-loop from k to ℓ. Define an infinite lasso path π as shown in Fig. 12: $\pi = (s_0, ..., s_{\ell-1},\ s_\ell, ...s_k,\ s_\ell, ..., s_k,\ ...)$. We construct a propositional formula ${}_\ell[\![f]\!]^0_k$ that holds iff f holds on π (in the usual LTL semantics).

Definition 11 (Original translation of LTL formula with a loop).

$$
\begin{aligned}
{}_\ell[\![\, p \,]\!]^i_k &:= p(s_i) \qquad \text{where } p \text{ is an atomic proposition} \\
{}_\ell[\![\, \neg p \,]\!]^i_k &:= \neg p(s_i) \qquad \text{where } p \text{ is an atomic proposition} \\
{}_\ell[\![\, f \vee g \,]\!]^i_k &:= {}_\ell[\![\, f \,]\!]^i_k \vee {}_\ell[\![\, g \,]\!]^i_k \\
{}_\ell[\![\, f \wedge g \,]\!]^i_k &:= {}_\ell[\![\, f \,]\!]^i_k \wedge {}_\ell[\![\, g \,]\!]^i_k \\
{}_\ell[\![\, \mathbf{X}\, f \,]\!]^i_k &:= \begin{cases} {}_\ell[\![\, f \,]\!]^{i+1}_k & \text{if } i < k \\ {}_\ell[\![\, f \,]\!]^{\ell}_k & \text{if } i = k \end{cases} \\
{}_\ell[\![\, \mathbf{G}\, f \,]\!]^i_k &:= \bigwedge_{j=\min(i,\ell)}^{k} {}_\ell[\![\, f \,]\!]^j_k \\
{}_\ell[\![\, \mathbf{F}\, f \,]\!]^i_k &:= \bigvee_{j=\min(i,\ell)}^{k} {}_\ell[\![\, f \,]\!]^j_k \\
{}_\ell[\![\, f \ \mathbf{U}\ g \,]\!]^i_k &:= \underbrace{\bigvee_{j=i}^{k} \left({}_\ell[\![\, g \,]\!]^j_k \wedge \bigwedge_{n=i}^{j-1} {}_\ell[\![\, f \,]\!]^n_k \right)}_{\text{Similar to loop-free case}} \vee \underbrace{\bigvee_{j=\ell}^{i-1} \left({}_\ell[\![\, g \,]\!]^j_k \wedge \bigwedge_{n=i}^{k} {}_\ell[\![\, f \,]\!]^n_k \wedge \bigwedge_{n=\ell}^{j-1} {}_\ell[\![\, f \,]\!]^n_k \right)}_{\text{See Fig. 14}}
\end{aligned}
$$

The translation for ${}_\ell[\![\, f \ \mathbf{U}\ g \,]\!]^i_k$ deserves some explanation. The translation is a disjunction of two parts. The first part is similar to the loop-free case. The second part is illustrated in Fig. 14. It handles the case where f holds on all path suffixes from π^i through π^k, continues holding for π^ℓ through π^{j-1}, and then g holds on π^j. (Note that $\pi^{k+1} = \pi^\ell$, since π has infinite length.)

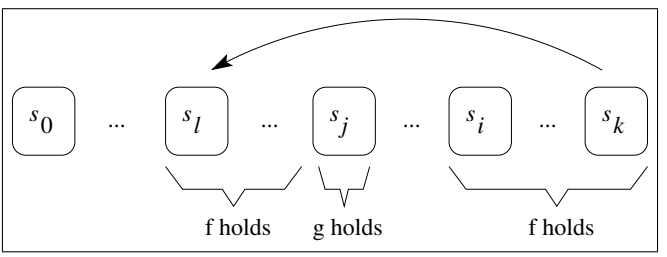

Fig. 14: Translation of ${}_\ell[\![\, f \ \mathbf{U}\ g \,]\!]^i_k$ for a path prefix with a back-loop.

Having defined the translation for paths both with and without back-loops, we are now almost ready to define the final translation into SAT. But first we need two auxiliary definitions. We define ${}_\ell L_k$ to be true iff there exists a transition from s_k to s_ℓ, and we define L_k to be true if there exists any possible back-loop in $(s_0, ..., s_k)$.

Definition 12 (Loop Condition). *For $l \leq k$, let ${}_\ell L_k := R(s_k, s_\ell)$, and let $L_k := \bigvee_{\ell=0}^{k} {}_\ell L_k$.*

Now we are ready to state the final translation into SAT, which we denote by "$[\![M, f]\!]_k$":

$$[\![M, f]\!]_k := \underbrace{[\![M]\!]_k}_{\text{valid prefix}} \wedge \Big(\underbrace{(\neg L_k \wedge [\![f]\!]_k^0)}_{\text{loop-free case}} \vee \underbrace{\bigvee_{\ell=0}^{k} ({}_\ell L_k \wedge {}_\ell[\![f]\!]_k^0)}_{\text{case with loop}} \Big) .$$

Theorem 2. Given a LTL formula f, there exists a path π that satisfies f iff there exists a k such that $[\![M, f]\!]_k$ is satisfiable. Equivalently, $M \models \mathbf{A}\neg f$ iff $[\![M, f]\!]_k$ is unsatisfiable for all k.

6.4 Improved Encoding for General LTL Properties

The translations that we have given above in Definitions 10 and 11 are not the most efficient, although they have the benefit of being relatively straightforward. More efficient translations are given in [15, 16, 2]; these translations have the benefit of having size linear in k (the unrolling depth) for the **U** operator, compared to size cubic in k (or quadratic in k, if certain optimizations [6] are used) for the translations in Definitions 10 and 11.

We use the same formula $[\![M]\!]_k$ as the original encoding (defined in Equation 3 on page 22) to constrain the path to be a valid prefix. In addition, we define formulas for *loop constraints*, which are used to non-deterministically select at most one back-loop in the path prefix $(s_0, ..., s_k)$. We introduce $k+1$ fresh *loop selector variables*, $l_0, ..., l_k$, which determine which possible back-loop (if any) to select. If l_j is true (where $1 \leq j \leq k$), then we select a back-loop from k to j. The state s_{j-1} is constrained to be equal to the state s_k, and we consider an infinite path $\pi = (s_0, ..., s_{j-1},\ s_j, ..., s_k,\ s_j, ..., s_k,\ ...)$. If none of the loop selector variables are true, we use the bounded semantics (Definition 9 on page 22).

We introduce auxiliary variables InLoop_0 through InLoop_k, which will be constrained so that InLoop_i is true iff position i is in the loop part of the path. In other words, InLoop_i should be true iff there exist a position $j \leq i$ such that l_j is true. To ensure that at most one of $\{l_0, ..., l_k\}$ is true, we require that l_i must not be true if there exists an earlier position $j < i$ such that l_j is true. Let $|[\mathit{LoopConstraints}]|_k$ be the conjunction of the following formulas for $i \in \{1, ..., k\}$:

$$\begin{aligned} l_0 &\Leftrightarrow \mathsf{false} \\ l_i &\Rightarrow (s_{i-1} = s_k) \\ \text{InLoop}_0 &\Leftrightarrow \mathsf{false} \\ \text{InLoop}_i &\Leftrightarrow \text{InLoop}_{i-1} \vee l_i \\ \text{InLoop}_{i-1} &\Rightarrow \neg l_i \end{aligned}$$

In Fig. 15, we define a function $|[f]|_0$ that translates an LTL formula f into a Boolean formula that indicates whether the path prefix $(s_0, ..., s_k)$ is a witness for f. If none of the loop selector variables are true, then $|[f]|_{k+1}$ simplifies to false, in accord with the bounded semantics. If a single loop selector variable l_j is true, we consider an infinite path $\pi = (s_0, ..., s_{j-1},\ s_j, ..., s_k,\ s_j, ..., s_k,\ ...)$. Note that the infinite path suffix π^{k+1} is equal to π^j. Thus, the translation for $|[f]|_{k+1}$ simplifies to $|[f]|_j$, except in the case of the $\mathbf{U}$ operator.

For $|[f\ \mathbf{U}\ g]|_i$, we make two passes through the loop part of path prefix. On the first pass, we consider path suffixes π^i through π^k (see Fig. 16). If f holds true for all these path suffixes, but g never holds true, then we need to make a second pass and continue checking at the start of the back-loop (π^j). If we reach position k on the second pass without g ever being true, then we know that g is never true at any position in the loop, so $f\ \mathbf{U}\ g$ is false. The auxiliary definition $\langle\!\langle f\ \mathbf{U}\ g \rangle\!\rangle$ handles the second pass. The final encoding for Kripke structure M, LTL formula f, and bound k is given by $|[M, f, k]|$:

$$|[M, f, k]| \ = \ |[M]|_k \wedge |[\mathit{LoopConstraints}]|_k \wedge |[f]|_0$$

Formula	Translation for $i \leq k$	Translation for $i = k + 1$
$\|[p]\|_i$	$p(s_i)$	$\bigvee_{j=1}^{k} \left(l_j \wedge \|[p]\|_j\right)$
$\|[\neg p]\|_i$	$\neg p(s_i)$	$\bigvee_{j=1}^{k} \left(l_j \wedge \|[\neg p]\|_j\right)$
$\|[f \wedge g]\|_i$	$\|[f]\|_i \wedge \|[g]\|_i$	$\bigvee_{j=1}^{k} \left(l_j \wedge \|[f \wedge g]\|_j\right)$
$\|[f \vee g]\|_i$	$\|[f]\|_i \vee \|[g]\|_i$	$\bigvee_{j=1}^{k} \left(l_j \wedge \|[f \vee g]\|_j\right)$
$\|[\mathbf{X}\, f]\|_i$	$\|[f]\|_{i+1}$	$\bigvee_{j=1}^{k} \left(l_j \wedge \|[\mathbf{X}\, f]\|_j\right)$
$\|[f\ \mathbf{U}\ g]\|_i$	$\|[g]\|_i \vee \left(\|[f]\|_i \wedge \|[f\ \mathbf{U}\ g]\|_{i+1}\right)$	$\bigvee_{j=1}^{k} \left(l_j \wedge \langle\!\langle f\ \mathbf{U}\ g \rangle\!\rangle_j\right)$
$\langle\!\langle f\ \mathbf{U}\ g \rangle\!\rangle_i$	$\|[g]\|_i \vee \left(\|[f]\|_i \wedge \langle\!\langle f\ \mathbf{U}\ g \rangle\!\rangle_{i+1}\right)$	false

Fig. 15: Improved BMC Translation

More compact translations are possible using Quantified Boolean Formulas (QBF) [14], although such translations generally cannot be solved faster (with existing QBF solvers) than SAT translations.

6.5 SAT Solvers

In this Section we discuss Boolean satisfiability (SAT) solvers, the key tool in bounded model checking. We begin with a few basic definitions.

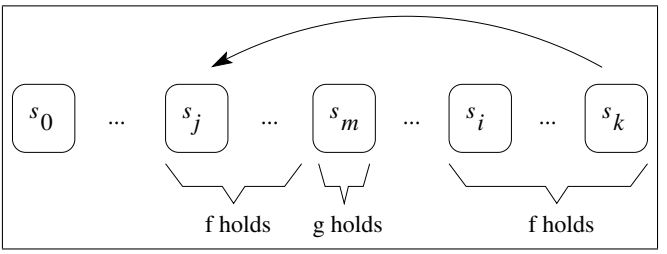

Fig. 16: Translation for a path prefix with a back-loop.

Definition 13. *A formula is* ***satisfiable*** *iff there exists an assignment to the variables of the formula that makes the formula true.*

Definition 14. *A formula is in* ***negation-normal form*** *(NNF) iff:*

- *all negations are directly in front of variables, and*
- *the only logical connectives are conjunction ("$\wedge$"), disjunction ("$\vee$"), and negation ("$\neg$").*

Definition 15. *A* ***literal*** *is a variable or its negation.*

Definition 16. *A formula is in* ***disjunctive normal form*** *(DNF) iff it is a disjunction of conjunctions of literals.*

For example, the below formula is in DNF:

$$(\ell_1 \wedge \ell_2 \wedge \ell_3) \vee (\ell_4 \wedge \ell_5 \wedge \ell_6) \vee (\ell_7 \wedge \ell_8 \wedge \ell_9) \ .$$

Note that every formula in DNF is also in NNF. A simple (but inefficient) way to convert an arbitrary formula ϕ to DNF is to make a truth table for ϕ. Every row in the truth table where ϕ is true corresponds to a conjunct in this DNF representation. For example, consider the truth table below.

x	y	z	$\phi(x, y, z)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

A DNF representation of the formula specified by the truth table is

$$(\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge \neg z) \ .$$

Definition 17. *A* ***clause*** *is a disjunction of literals.*

Definition 18. *A formula is in* ***conjunctive normal form*** *(CNF) iff it is a conjunction of clauses (disjunctions of literals).*

For example, the below formula is in CNF:

$$(\ell_1 \vee \ell_2 \vee \ell_3) \wedge (\ell_4 \vee \ell_5 \vee \ell_6) \wedge (\ell_7 \vee \ell_8 \vee \ell_9) \ .$$

Modern SAT solvers require their input to be in CNF. An arbitrary formula can be converted to CNF in a manner similar to the above-described method for DNF. However, the resulting CNF formula can be exponentially larger than the original formula. A better way of converting to CNF is to use the *Tseitin transformation* [24], which produces an *equisatisfiable* formula that is only linearly larger than the original formula. Two formulas are equisatisfiable when either they are both satisfiable or they are both unsatisfiable.

The Tseitin transformation introduces new variables to represent subformulas of the original formula. The technique is perhaps best illustrated by considering a formula as a combinational circuit. A new variable is introduced to represent the output of each logic gate in the circuit. Let g_{top} be the variable corresponding to the top-level gate (i.e., the gate whose output is the output of the circuit). We create the CNF formula by conjoining the unit clause (g_{top}) with clauses that relate each newly introduced gate variable to the inputs of the corresponding logic gate. For example, consider the circuit for $(A \vee (B \wedge C))$ represented in Fig. 17. To obtain the clauses that define g_1, we start with the equivalence

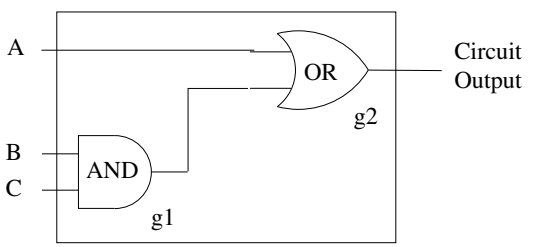

Fig. 17: Circuit representation of $(A \vee (B \wedge C))$

$g_1 \Leftrightarrow (B \wedge C)$. Then we break the equivalence into two implications, and then we convert the implications to clauses using the fact that $(X \Rightarrow Y)$ is logically equivalent to $(\neg X \vee Y)$. Therefore, we have

$$\begin{aligned}(g_1 \Leftrightarrow (B \wedge C)) &= (g_1 \Rightarrow (B \wedge C)) \wedge ((B \wedge C) \Rightarrow g_1) \\ &= (g_1 \Rightarrow B) \wedge (g_1 \Rightarrow C) \wedge ((B \wedge C) \Rightarrow g_1) \\ &= (\neg g_1 \vee B) \wedge (\neg g_1 \vee C) \wedge (\neg B \vee \neg C \vee g_1) \ .\end{aligned}$$

Note that since the Tseitin transformation introduces new variables, the resulting formula is not strictly equivalent to the original formula. Instead, it is equisatisfiable with the original formula.

We now briefly describe the main principles followed by SAT solvers. A CNF formula is represented by a set of clauses, and a clause is represented by a set of literals.

Definition 19. *A **unit clause** is a clause that contains exactly one literal.*

Almost all modern SAT solvers use a variant of the DPLL algorithm [12]. This algorithm uses a backtracking search. The solver picks a variable in the input formula and decides a value for it. It then performs *unit propagation*: If the formula has a clause with exactly one literal, then the solver assigns that literal true and simplifies the formula under that assignment. Unit propagation is repeated until there are no more unit clauses. If a satisfying assignment is discovered, the solver returns `true`. If a falsifying assignment is discovered, the solver backtracks, undoing its decisions. High-level pseudocode of the DPLL algorithm is shown in Fig. 18.

Great improvements to SAT solvers have been made beginning in the mid-1990s. GRASP [12] introduced a powerful form of conflict analysis that enables (1) non-chronological backtracking, allowing the solver to avoid considering unfruitful assignments, and (2) learning of additional implied clauses, which enables to solver to discover more implied literals via unit propagation. When unit propagation forces a literal to be assigned a certain value, GRASP records the set of literals responsible. When a conflict is discovered, GRASP uses this information to derive a new clause. The learned clause is logically redundant, but it enables unit propagation to be more effective.

Another major breakthrough is the *two watched literals* scheme introduced by Chaff [19]. SAT solvers spend most of their time doing unit propagation, and the watched-literals scheme makes unit propagation significantly more efficient.

```
function Solve(φ) {
    φ := Propagate(φ);
    if (φ = true) {return true;}
    if (φ = false) {return false;}
    x := (pick a variable in φ);
    return (Solve(φ[x/true]) or Solve(φ[x/false]));
}
```

Fig. 18: DPLL pseudocode; $\phi[x/c]$ denotes syntactic substitution of c for x in ϕ.

7 Conclusion

The state explosion problem has been, and is likely to remain, the main challenge faced by model checking. Of the techniques developed to combat the state explosion problem, bounded model checking is one of the most successful. In practice,

bounded model checking can often find counterexamples in circuits with thousands of latches and inputs. Armin Biere reported an example in which the circuit had 9510 latches and 9499 inputs. This resulted in a propositional formula with 4 million variables and 12 million clauses. The shortest bug of length 37 was found in only 69 seconds! Many others have reported similar results.

There are many directions for future research, including software model checking, hybrid-systems model checking, compositional model checking, statistical model checking, combining model checking with theorem proving, and scaling up even more!

References

1. A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs. In *CAV*, 1999.
2. A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan. Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science*, 2(5), 2006.
3. R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, 1986.
4. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 13(4):401–424, 1994.
5. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2):142–170, 1992.
6. A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In *VMCAI*, 2002.
7. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1981. Springer-Verlag.
8. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In *POPL*, pages 117–126, 1983.
9. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic Model Checking. *J. ACM*, 50(5):752–794, 2003. Originally presented at CAV'00.
10. E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
11. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.
12. M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
13. M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based unbounded symbolic Model Checking using circuit cofactoring. In *International conference on Computer-aided design (ICCAD'04)*, pages 510–517, 2004.
14. T. Jussila and A. Biere. Compressing BMC Encodings with QBF. *Electr. Notes Theor. Comput. Sci.*, 174(3):45–56, 2007.
15. T. Latvala, A. Biere, K. Heljanko, and T. A. Junttila. Simple Bounded LTL Model Checking. In *FMCAD*, pages 186–200, 2004.

16. T. Latvala, A. Biere, K. Heljanko, and T. A. Junttila. Simple Is Better: Efficient Bounded Model Checking for Past LTL. In *VMCAI*, pages 380–395, 2005.
17. K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 250–264, 2002.
18. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *Computer-Aided Verification (CAV'03)*, LNCS 2725, pages 1–13, 2003.
19. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535. ACM, 2001.
20. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
21. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
22. M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, LNCS 1954, pages 108–125, 2000.
23. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.
24. G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, Part II, ed. A.O. Slisenko, 1968.