

## Chapter 2

# SYMBOLIC MODEL CHECKING

Model checking [CE81, QS81] is an algorithmic method for proving that a digital system satisfies a user-defined specification. Both the system and the specification must be formally specified: The model of the system must have a finite number of states; the specification, or property, is often expressed in temporal logics. In the model checking literature, the model and the property are often represented by the Kripke structure and a temporal logic formula, respectively.

Given a model  $K$  and a property  $\phi$ , model checking is used to check whether  $K$  models  $\phi$ , denoted by  $K \models \phi$ . For properties specified in Computational Tree Logic (CTL [CE81, EH83]), the model checking problem can be solved by a set of least and/or greatest fixpoint computations [CES86]. For properties specified in Linear Time Logic (LTL [Pnu77]), model checking is often transformed into language emptiness checking in a generalized Büchi automaton. In this *automata-theoretic* approach [VW86], the negation of the given LTL formula is encoded into a Büchi automaton, which is then composed with the model. The LTL model checking problem is then decided by checking the language of the composed system—the model satisfies the property if and only if the language of the composed system is empty. Therefore, the underlying LTL model checking algorithms are usually variants of algorithms for computing Strongly-Connected Components (SCCs).

In this chapter, we first introduce the basic concepts and notations commonly used in model checking. We then review some of the fundamental algorithms in symbolic model checking, which includes BDD based symbolic fixpoint computation, SCC hull and SCC enumeration algorithms, SAT and bounded model checking, and iterative abstraction refinement.

## 2.1 Finite State Model

In model checking, we deal with a formal model of the given digital system, known as the Kripke structure. A Kripke structure is an annotated finite-state transition graph.

**DEFINITION 2.1** *A Kripke structure is a 5-tuple*

$$K = \langle S, S_0, T, A, \Lambda \rangle ,$$

where  $S$  is a finite set of states,  $S_0 \subseteq S$  is the set of initial states,  $T \subseteq S \times S$  is the transition relation,  $A$  is a finite alphabet for which a set  $P$  of atomic propositions is given and  $A = 2^P$ , and  $\Lambda : S \rightarrow A$  is the labeling function.

We further require that the transition relation of a Kripke structure be complete; that is, every state has at least one successor. With this assumption, we can extend any finite state path in the state transition graph into an infinite one.

As the standard representation of models in the model checking literature, the Kripke structure has its origin in modal logic, the generalization of temporal logic. In modal logic, a certain formula is interpreted with respect to a state inside a universe, a domain of discourse, and a relation establishing how the validity of a predicate changes from state to state. Temporal logic is a special case of modal logic that allows us to reason about how predicates evolve over time. In temporal logic model checking, a node or state of the Kripke structure represents the “state” of the given system at a certain time, and the change from state to state represents a time change.

From an engineer’s point of view, the Kripke structure is nothing but a labeled finite state machine (FSM). The additional features, i.e., the finite alphabet and a labeling function from states to sets of atomic propositions, make it possible to specify simple propositional properties on the finite state machine. These propositional properties, combined with some temporal operators, allow us to specify properties like “ $\neg$ abort holds on all the states reachable from the initial states” or “from a state labeled *req* we will eventually reach a state labeled *ack*.” We will introduce temporal logic operators in the next section. Now let us focus on propositional properties and take a look at the example FSM at the right-hand side of Figure 2.1.

The FSM in Figure 2.1 has four states, among which three are reachable from the single initial state  $a$ . Propositions  $p$  and  $q$  belong to the finite alphabet. With the labeling function and initial predicate indicated in Figure 2.1, the finite state machine is augmented into a Kripke

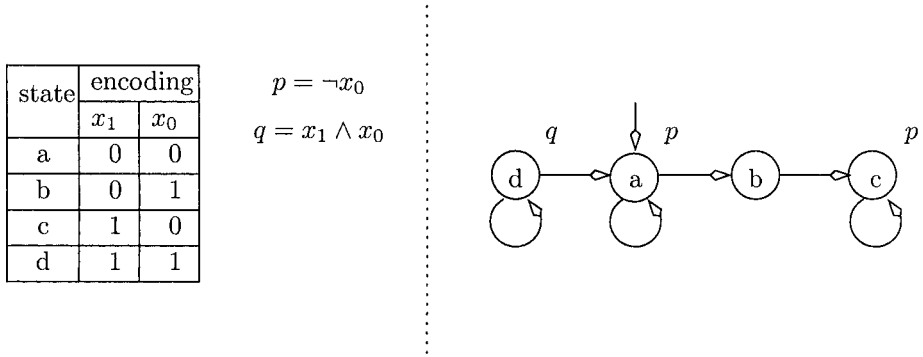


Figure 2.1. An example of the Kripke structure.

structure defined as follows:

$$\begin{array}{ll}
 S &= \{a, b, c, d\} \\
 S_0 &= \{a\} \\
 T &= \{(a, a), (a, b), (b, c), (c, c), (d, d), (d, a)\} \\
 P &= \{p, q\}
 \end{array}
 \qquad
 \begin{array}{ll}
 \Lambda(a) &= \{p\} \\
 \Lambda(b) &= \{\} \\
 \Lambda(c) &= \{p\} \\
 \Lambda(d) &= \{q\}
 \end{array}$$

Given a sequential circuit, the construction of the finite state machine from the system description is straightforward. A digital circuit is often defined as an entity with memory elements (latches and flip-flops), combinational logic gates, input signals, and internal wires. The transition functions of the memory elements are defined in terms of the current values of these memory elements and the input signals.

Figure 2.2 gives an example circuit, in which we use the variables  $x_1$  and  $x_0$  to represent the outputs of the two registers, and variables  $y_1$  and  $y_0$  to represent their data inputs. Note that after a clock cycle, the values of  $y_1$  and  $y_0$  will be propagated to the register outputs. Therefore, we often call  $x_1$  and  $x_0$  the present-state (or current-state) variables, and call  $y_1$  and  $y_0$  the next-state variables. In this example, we use the variable  $w_0$  to represent the value of a primary input signal.

States in the corresponding FSM are mapped to the different valuations of the set of memory elements. Edges in the state transition graph correspond to the changes of states among different clock cycles. For the example in Figure 2.2, we can write out the transition functions of the

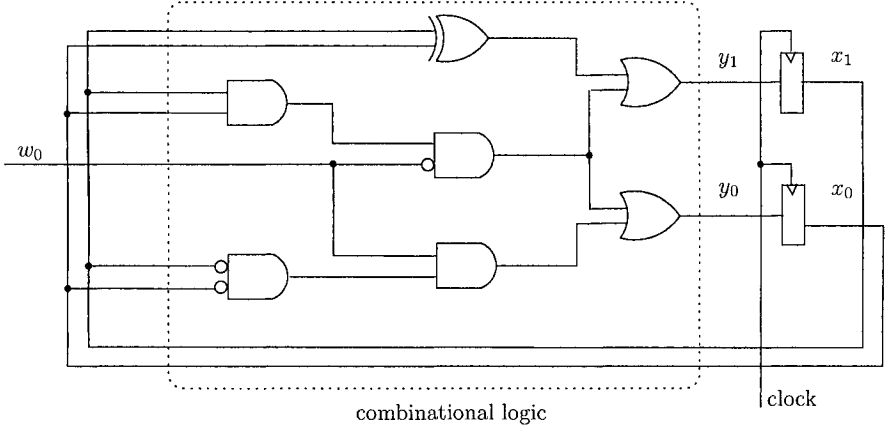


Figure 2.2. A sequential circuit example.

two registers as

$$\begin{aligned} y_1 &: \neg x_1 \wedge x_0 \vee x_1 \wedge \neg x_0 \vee x_1 \wedge x_0 \wedge \neg w_0 \\ y_0 &: \neg x_1 \wedge \neg x_0 \wedge w_0 \vee x_1 \wedge x_0 \wedge \neg w_0 \end{aligned}$$

Given the values of present-state variables and the input signal, the values of next-state variables are determined by their transition functions. When the current values of the two registers are  $(x_1 = 0, x_0 = 0)$ , for instance, their values at the next clock cycle will be  $(y_1 = 0, y_0 = 0)$  for  $w_0 = 0$ , and  $(y_1 = 0, y_0 = 1)$  for  $w_0 = 1$ . If we use the state encoding scheme and labeling functions described on the left-hand side of Figure 2.1, we will get the right-hand side Kripke structure in the same figure.

Since the number of memory elements in a sequential circuit is finite, there are only a finite number of states. However, There is a well-known *state explosion* problem. The total number of states in the FSM can be as large as  $2^n$  for a system with  $n$  binary state variables. Due to its exponential dependence on the number of state variables, the number of states of the model can be extremely large even for a moderate-size system.

Some digital systems may have an infinite number of states. Software with recursive function calls and unbounded data structures, for instance, fall into this category. Other examples include timed systems and hybrid systems [ACH<sup>+</sup>95, AH96], in which the state variables can be of unbounded integer or even real type. Since model checking requires the Kripke structure to be finite-state, before we can apply model

checking, a certain degree of abstraction is needed to extract suitable verification models from these systems. In general, abstraction used for this purpose is either under-approximation or over-approximation. The process of mapping an infinite state space into a finite state space, by itself, is an important research topic, and is beyond the scope of this book. In the sequel, we assume that the finite state model of a given system, or the Kripke structure, is already available.

## 2.2 Temporal Logic Property

Propositional logic is the basis for specifying properties. A *proposition* is a declarative sentence about the Kripke structure that is either true or false. Propositions are represented by a set of propositional variables  $p, q, \dots$  plus the truth values **true** and **false**. A formula consisting of a propositional variable is called an *atomic proposition*. The evaluation of an atomic proposition maps to a set of states in the Kripke structure.

Propositional logic formulae are defined in terms of atomic propositions with the common logical connectives.

**DEFINITION 2.2** *A propositional logic formula is defined as follows:*

- *atomic propositions are propositional formulae;*
- *if  $\phi$  is a propositional formula, then  $\neg\phi$  is a propositional formula;*
- *if  $\phi$  and  $\psi$  are propositional formulae, then  $\phi \wedge \psi$ ,  $\phi \vee \psi$ ,  $\phi \rightarrow \psi$ ,  $\phi \leftrightarrow \psi$  are propositional formulae.*

In the set of logical connectives, the unary operator negation ( $\neg$ ) and the binary operation logical AND ( $\wedge$ ) constitute a minimal subset that is sufficient for defining propositional logic. Besides  $\wedge$ , there are 15 other binary logical connectives; however, all of them can be expressed in terms of  $\neg$  and  $\wedge$ . For example, under the De Morgan's law the formula  $\phi \vee \psi$  can be rewritten into  $\neg(\neg\phi \wedge \neg\psi)$ . The “implies” operator  $\rightarrow$  means “only if”, and therefore  $\phi \rightarrow \psi$  is equivalent to  $\neg\phi \vee \psi$ . Similarly, the formula  $\phi \leftrightarrow \psi$  is equivalent to  $\neg\phi \wedge \neg\psi \vee \phi \wedge \psi$ .

Propositional logic is incapable of reasoning about the evolution of valuations over time. When the truth of a property depends on not only the present valuation, but also on the valuations in the past or in the future, we need temporal logics. The most common temporal logics to express system properties are Computational Tree Logic (CTL) and Linear Time Temporal Logic (LTL). CTL and LTL are subsets of the more general CTL\*. In this book, we will focus on Linear Time Temporal Logic, but we will also briefly describe the Computational Tree Logic, since some of its operators will be used in our discussion of model checking algorithms.

There are two very different ways of modeling time in temporal logics. The linear time model assumes that each time instance has exactly one successor; the branching time model, on the other hand, allows several successors for each time instance. LTL is based on the linear time model. LTL formulae specify properties about the future of each individual execution trace such as the condition `ack` will eventually be true, or that the condition `busy` will be true until another condition `done` becomes true. Logics based on the branching time model, such as CTL, deal with all possible execution traces. CTL formulae can specify properties such as that if the condition `reset` is true then on all paths the condition `reset_done` will eventually be true.

LTL formulae are defined in terms of atomic propositions, the usual logic connectives, as well as linear time temporal operators. The two basic temporal operators in LTL are  $X$  and  $U$ , called *next* and *until*, respectively. The first operator is unary and the second is binary. The formula  $X\phi$  means that  $\phi$  holds at the next point of time. The formula  $\phi U \psi$  means that  $\phi$  has to hold until  $\psi$  becomes true, and  $\psi$  will eventually become true.

**DEFINITION 2.3** *A Linear Time Temporal Logic (LTL) formula is defined recursively as follows:*

- *atomic propositions are LTL formulae;*
- *if  $\phi$  and  $\psi$  are LTL formulae, so are  $\neg\phi$ ,  $\phi \wedge \psi$ , and  $\phi \vee \psi$ ;*
- *if  $\phi$  and  $\psi$  are LTL formulae, so are  $X\phi$  and  $\phi U \psi$ ;*

Besides  $X$  and  $U$ , there are other temporal operators including  $G$  for *globally*,  $F$  for *finally*, and  $R$  for *release*. The formula  $G\phi$  means that  $\phi$  has to hold forever. The formula  $F\phi$  means that  $\phi$  will eventually be true. The formula  $\phi R \psi$  means that  $\psi$  remains true before the first time  $\phi$  becomes true (or forever if  $\phi$  remains false). These three temporal operators can be expressed in terms of the two basic ones:

$$\begin{aligned} F\phi &= \text{true} U \phi \\ G\phi &= \neg F \neg\phi \\ \phi R \psi &= \neg(\neg\psi U \neg\phi) \end{aligned}$$

The semantics of LTL formulae are defined for an infinite path  $\pi = (s_0, s_1, \dots)$  of the Kripke structure, where  $s_i \in S$  is a state,  $s_0$  is an initial state, and  $T(s_i, s_{i+1})$  evaluates to true for all  $i \geq 0$ . The suffix of  $\pi$  starting from the state  $s_i$  is represented by  $\pi^i$ . We use  $K, \pi^i \models \phi$  to represent the fact that  $\phi$  holds in a suffix of path  $\pi$  of the Kripke

structure  $K$ . The property  $\phi$  holds for the entire path  $\pi$  if and only if  $K, \pi^0 \models \phi$ . When the context is clear, we will omit  $K$  and rewrite  $K, \pi^i \models \phi$  into  $\pi^i \models \phi$ . The semantics of LTL formulae are defined recursively as follows:

|                                   |   |
|-----------------------------------|---|
| $\pi \models \text{true}$         | always holds  |
| $\pi \models \varphi$             | iff $\pi^0 \models \varphi$   |
| $\pi \models \neg\varphi$         | iff $\pi \not\models \varphi$   |
| $\pi \models \varphi \wedge \psi$ | iff $\pi \models \varphi$ and $\pi \models \psi$  |
| $\pi \models X\varphi$            | iff $\pi^1 \models \varphi$   |
| $\pi \models \varphi U \psi$      | iff $\exists i \geq 0$ such that $\pi^i \models \psi$ and for all $0 \leq j < i$ ,<br>$\pi^j \models \varphi$   |
| $\pi \models \varphi R \psi$      | iff for all $i \geq 0$ , $\pi^i \models \psi$ ; or $\exists j \geq 0$ such that<br>$\pi^j \models \varphi$ and for all $0 \leq i \leq j$ , $\pi^i \models \psi$ |

The Kripke structure  $K$  satisfies an LTL formula  $\phi$  if and only if all paths from the initial states do. This means that all LTL properties are universal properties in the sense that we can add the path quantifier  $A$  as a prefix without changing the meaning of the properties. That is,  $K \models \phi$  is equivalent to  $K \models A\phi$ , where the path quantifier  $A$  means  $\phi$  holds for all computation paths. Another path quantifier is  $E$ , which stands for *there exists a computation path*.  $E$  is not used in LTL, but both  $A$  and  $E$  are used in CTL.

An LTL formula is in the normal form if negation appears only in front of propositional formulae. For instance, the formula  $F\neg Fp$  is not in the normal form since negation is ahead of the temporal operator  $F$ ; on the other hand, the equivalent formula  $FG\neg p$  is in the normal form. We can always rewrite an LTL formula into normal form by pushing negation inside temporal operators. The following rules can be applied during the rewriting:

$$\begin{aligned}
 Xp &= \neg X\neg p \\
 Gp &= \neg F\neg p \\
 p U q &= \neg(\neg q R \neg p) \\
 Fp &= \text{true} U p
 \end{aligned}$$

Since an LTL formula  $\phi$  is a universal property and is equivalent to  $A\phi$ , the negation of  $\phi$  should be the existential property  $E\neg\phi$ .

The two path quantifiers are an integral part of Computational Tree Logic (CTL), and are used explicitly to specify properties related to execution traces in the computation tree structure.  $A$  (for all computation paths) specifies that all paths starting from a given state satisfy a property;  $E$  (for some computation paths) specifies that some of these paths satisfy a property.

DEFINITION 2.4 *A Computational Tree Logic (CTL) formula is defined recursively as follows:*

- *atomic propositions are CTL formulae;*
- *if  $\varphi$  and  $\psi$  are CTL formulae, then  $\neg\varphi$ ,  $\varphi \wedge \psi$ , and  $\varphi \vee \psi$  are CTL formulae;*
- *if  $\varphi$  and  $\psi$  are CTL formulae, then  $\text{EX } \varphi$ ,  $\text{E } \psi \text{ U } \varphi$ , and  $\text{EG } \varphi$  are CTL formulae.*

A CTL formula is in the normal form if negation appears only in front of propositional formulae. Formula  $\neg \text{AX } p$  is not in the normal form since negation is ahead of the temporal operator AX; on the other hand, the equivalent formula  $\text{EX } \neg p$  is in the normal form. We can always rewrite a CTL formula into normal form by pushing negation inside temporal operators. The following rewriting rules can be applied during normalization:

$$\begin{aligned}
 \text{AX } p &= \neg \text{EX } \neg p \\
 \text{AG } p &= \neg \text{EF } \neg p \\
 p \text{ U } q &= \neg(\text{E } \neg q \text{ U } \neg p \wedge \neg q) \wedge \neg \text{EG } \neg q \\
 \text{AF } p &= \text{A true U } p \\
 \text{EF } p &= \text{E true U } p
 \end{aligned}$$

Many interesting properties in practice can be expressed in both LTL and CTL. However, there are also properties that can be expressed in one but not the other. The difference between an LTL formula and a CTL formula can be very subtle. For instance, the LTL formula  $\text{FG } p$  holds in the Kripke structure in Figure 2.3, but the CTL formula  $\text{AF AG } p$  fails. (In the Kripke structure,  $p$  and  $q$  are state labels.)

The reason is that the LTL property is related to the individual paths, and on any infinite path of the given Kripke structure we can reach the state  $c$  from which  $p$  will hold forever. The CTL formula  $\text{AF AG } p$ , on the other hand, requires that on all paths from the state  $a$  we can reach a state satisfying  $\text{AG } p$ . Note that the only state satisfying  $\text{AG } p$  is the state  $c$ ; however, the Kripke structure does not satisfy  $\text{AF}\{c\}$ —as shown in the right-hand side of the figure, the left most path of the computation tree is a counterexample. On this particular path, we can stay in the state  $a$  while reserving the possibility of going to the state  $b$  (where  $p$  does not hold). Therefore,  $\text{FG } p$  and  $\text{AF AG } p$  represent two very similar but different properties.

The above example shows that LTL and CTL have different expressing powers. Some LTL properties, like  $\text{FG } p$ , cannot be expressed in



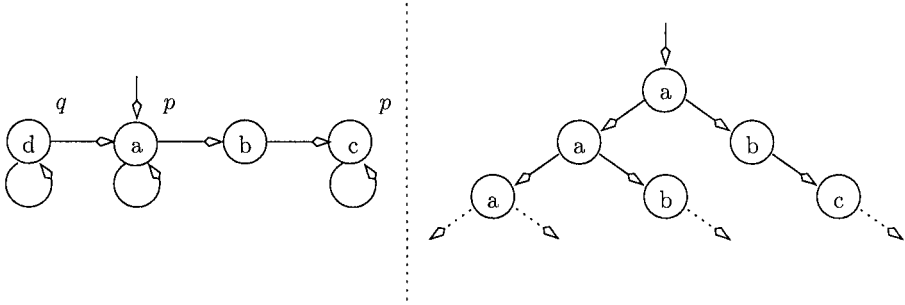


Figure 2.3. A Kripke structure and its computation tree.

CTL. There are also CTL properties that cannot be expressed in LTL; an example in this category would be  $\text{AG EF } p$ . Both LTL and CTL are strict subsets of the more general CTL\* logic [EH83, EL87]. The relationship among LTL, CTL, and CTL\* is given in Figure 2.4. In this book, we focus primarily on LTL model checking. Readers who are interested in CTL model checking are referred to [CES86, McM94] or the book [CGP99].

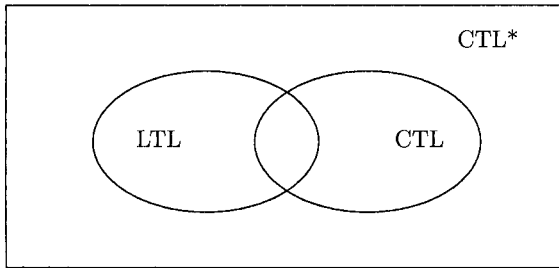


Figure 2.4. The relationship among LTL, CTL, and CTL\*.

We have used the term *universal property* during previous discussions. Now we give a formal definition of universal and existential properties.

**DEFINITION 2.5** *A property  $\phi$  is a universal property if removing edges from the state transition graph of the Kripke structure does not reduce the set of states satisfying  $\phi$ . A property  $\psi$  is an existential property if adding edges into the state transition graph of the Kripke structure does not reduce the set of states satisfying  $\psi$ .*

It follows that all LTL properties and ACTL (the universal fragment of CTL) properties are universal. The existential fragment of CTL, or ECTL, is existential. For the propositional  $\mu$ -calculus formulae, those that do not use EX and EY in their normal forms are universal.

Temporal logic properties can also be classified into the following two categories: *safety* properties and *liveness* properties. The notion of safety and liveness was introduced first by Lamport [Lam77]. Alpern and Schneider [AS85] later gave a formal definition of both safety and liveness properties. Informally, a safety property states that something bad will not happen during a system execution. Liveness properties are dual to safety properties, expressing that eventually something good must happen. The distinction of safety and liveness properties was originally motivated by the different techniques for proving them.

We can think of a property as a set of execution sequences, each of which is an infinite sequence of states of the Kripke structure. A property is called a safety property if and only if each execution violating the property has a finite prefix violating that property. In other words, a finite prefix of an execution violating the property (bad thing) is irremediable no matter how the prefix is extended to an infinite path. Safety properties can be falsified in a finite initial part of the execution, although proving them requires the traversal of the entire set of reachable states. The invariant property  $Gp$  or  $AGp$ , which states that the propositional formula  $p$  always holds, is a safety property. Other safety properties include mutual exclusion, deadlock freedom, etc.

A property is a liveness property if and only if it contains at least one good continuation for every finite prefix. This corresponds to the intuition that it is still possible for the property to hold (good thing to happen) after any finite execution. Liveness properties do not have a finite counterexample, and therefore in principle cannot be falsified after a finite number of execution steps. An example of liveness property is  $G(p \rightarrow Fq)$ , which states that whenever the propositional formula  $p$  is true, the propositional formula  $q$  must become true at some future cycle although there is no upper limit on the time by which  $q$  is required to become true. Other liveness properties include accessibility, absence of starvation, etc.

### 2.3 Generalized Büchi Automaton

An LTL formula  $\phi$  always corresponds to a Büchi automaton  $\mathcal{A}_\phi$  that recognizes all its satisfying infinite paths. In other words, the Büchi automaton  $\mathcal{A}_\phi$  contains all the logic models of the formula  $\phi$ . If we consider the Kripke structure as a language generator and the Büchi automaton  $\mathcal{A}_\phi$  as a language recognizer, then we have  $K \models \phi$  if and only

if all infinite words generated by  $K$  are accepted by  $\mathcal{A}_\phi$ . Therefore, the LTL model checking problem can be translated to  $\omega$ -regular language containment checking. Since checking language containment between two Büchi automata in general is PSPACE-complete [Kur94], it follows that LTL model checking is PSPACE-complete.

In practice, however, LTL model checking is often translated into language emptiness checking in a generalized Büchi automaton. This *automata-theoretic* approach [VW86] consists of the following three steps:

- 1 we negate the given property  $\phi$  and translate it into a Büchi automaton  $\mathcal{A}_{\neg\phi}$ , which accepts all the infinite paths that do not satisfy  $\phi$ ;
- 2 we then compose the model  $K$  and the property automaton  $\mathcal{A}_{\neg\phi}$  together. The system produced by parallel composition, denoted by  $(K||\mathcal{A}_{\neg\phi})$ , consists of only those infinite paths of  $K$  that are accepted by  $\mathcal{A}_{\neg\phi}$ ;
- 3 finally, we check whether the language of the composed system is empty.

If the language is empty, then  $K \models \phi$  since no infinite path in  $K$  is accepted by  $\mathcal{A}_{\neg\phi}$ . If the language is not empty, any accepting run in the composed system serves as a counterexample to  $K \models \phi$ .

LTL model checking via language emptiness has the same worst-case complexity bound as the language containment based approach, which is linear in the number of states of the model, but exponential in the length of the LTL formula. The exponential blow-up comes from the translation from LTL formulae to Büchi automata. However, this is often acceptable in practice, because user specified LTL formulae are usually small compared to the size of the model.

In the automata-theoretic approach, we can use the labeled generalized Büchi automata as a unified representation for the model  $K$ , the property automaton  $\mathcal{A}_{\neg\phi}$ , as well as the composed system  $(K||\mathcal{A}_{\neg\phi})$ . A labeled generalized Büchi automaton is simply a Kripke structure augmented by a set of acceptance conditions. In other words, we can view the model  $K$  as a special case of the labeled generalized Büchi automaton whose only acceptance condition is satisfied by all paths.

**DEFINITION 2.6** *A labeled generalized Büchi automaton is a six-tuple*

$$\mathcal{A} = \langle S, S_0, T, A, \Lambda, \mathcal{F} \rangle ,$$

where  $S$  is the finite set of states,  $S_0 \subseteq S$  is the set of initial states,  $T \subseteq S \times S$  is the transition relation,  $A$  is a finite alphabet for which a

set  $P$  of atomic propositions is given and  $A = 2^P$ ,  $\Lambda : S \rightarrow A$  is the labeling function, and  $\mathcal{F} \subseteq 2^S$  is the set of acceptance conditions.

A run of  $\mathcal{A}$  is an infinite sequence  $\rho = s_0, s_1, \dots$  over  $S$ , such that  $s_0 \in S_0$  and for all  $i \geq 0$ ,  $(s_i, s_{i+1}) \in T$ . A run  $\rho$  is *accepting* or *fair* if for each fair set  $F_i \in \mathcal{F}$ , there exists  $s_j \in F_i$  that appears infinitely often in  $\rho$ . The automaton accepts an infinite word  $\sigma = \sigma_0, \sigma_1, \dots$  in  $A^\omega$  if there exists an accepting run  $\rho$  such that for all  $i \geq 0$ ,  $\sigma_i \in \Lambda(\rho_i)$ . The language of  $\mathcal{A}$ , denoted by  $\mathcal{L}(\mathcal{A})$ , is the subset of  $A^\omega$  accepted by  $\mathcal{A}$ . Note that the language of  $\mathcal{A}$  is nonempty if and only if  $\mathcal{A}$  contains a reachable fair cycle—a cycle that is reachable from an initial state and intersects with all the fair sets.

We have defined the automata with labels on the states, not on the edges. The automata are called generalized Büchi automata because multiple acceptance conditions are possible. A state  $s$  is *complete* if for every  $a \in A$ , there is a successor  $s'$  of  $s$  such that  $a \in \Lambda(s')$ . A set of states, or an automaton, is complete if all of its states are. In a complete automaton, any finite state path can be extended into an infinite run. In the sequel all automata are assumed to be complete.

We define the concrete system  $\mathcal{A}$  as the synchronous (or parallel) composition of a set of submodules. Composing a subset of these submodules gives us an over-approximated abstract model  $\mathcal{A}'$ . In symbolic algorithms,  $\mathcal{A}$  and  $\mathcal{A}'$ , as well as the submodules, are all defined over the same state space and agree on the state labels. Communication among submodules then proceeds through the common state space, and composition is characterized by the intersection of the transition relations.

**DEFINITION 2.7** *The composition  $\mathcal{A}_1 \parallel \mathcal{A}_2$  of two Büchi automata  $\mathcal{A}_1$  and  $\mathcal{A}_2$ , where*

$$\begin{aligned}\mathcal{A}_1 &= \langle S, S_{01}, T_1, A, \Lambda, \mathcal{F}_1 \rangle, \\ \mathcal{A}_2 &= \langle S, S_{02}, T_2, A, \Lambda, \mathcal{F}_2 \rangle,\end{aligned}$$

*is a Büchi automaton  $\mathcal{A} = \langle S, S_0, T, A, \Lambda, \mathcal{F} \rangle$  such that,  $S_0 = S_{01} \cap S_{02}$ ,  $T = T_1 \cap T_2$ , and  $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ .*

In Figure 2.5, we give an example to show how the automata-theoretic approach for LTL model checking works. The model or Kripke structure in this example corresponds to the circuit in Figure 2.2. We are interested in checking the LTL property  $\phi = \text{FG } p$ ; that is, eventually we will reach a point from which the propositional formula  $p$  holds for ever.

First, we create the property automaton  $\mathcal{A}_{\neg\phi}$  that admits all runs satisfying the negation of  $\phi$ , which is  $\neg\phi$  or  $\text{GF } \neg p$ . Runs satisfying  $\text{GF } \neg p$  must visit states labeled  $\neg p$  infinitely often. There are algorithms to

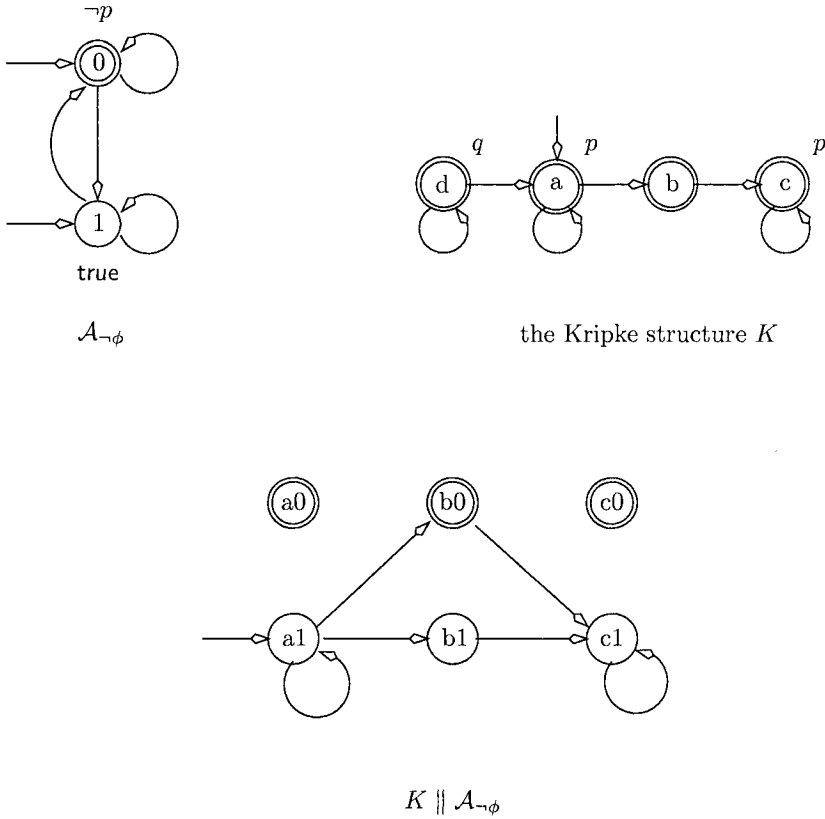


Figure 2.5. An LTL model checking example.

translate a general LTL formula to a Büchi automaton; readers who are interested in this subject are referred to [VW86, GPVW95, SB00]. Since our example is simple enough, we do not need to go through the detailed translation algorithm to convince ourselves that the automaton in Figure 2.5 indeed corresponds to  $\text{GF } \neg p$ . In Figure 2.5, states satisfying the acceptance condition are represented as double circles. The property automaton has only one acceptance condition  $\{0\}$ , meaning that any accepting run has to go through the state 0 infinitely often. We assume that all states in the Kripke structure  $K$  are accepting; that is, the acceptance condition of  $K$  is  $\{a, b, c, d\}$ .

After composing the property automaton with the model, we have the composed system at the bottom of Figure 2.5, whose acceptance condition is  $\{a0, b0, c0\}$ . Note that in the creation of  $K \parallel \mathcal{A}_{\neg\phi}$  we have used the parallel composition; that is, only transitions that are allowed

by both parents are retained. The final acceptance condition is also the union of those of both parents. Since the one for  $K$  consists of the entire state space, it is omitted. Finally, we check language emptiness in the composed system by searching for a run that goes through some states in the fair set  $\{a0, b0, c0\}$  infinitely often. It is clear that the language of that system is empty, because no run visits any of these states infinitely often. Therefore, the property  $\phi$  holds in  $K$ .

Whether the language of a Büchi automaton is empty can be decided by evaluating the temporal logic property  $\text{EG}_{\text{fair}} \text{true}$  on the automaton. In other words, the language of the composed system is empty if and only if no initial state of the composed system satisfies  $\text{EG}_{\text{fair}} \text{true}$ . The property is an existential CTL formula augmented with a set of Büchi fairness constraints; for our running example,  $\text{fair} = \{\{a0, b0, c0\}\}$ . In a run satisfying this property, a state in every  $F_i \in \mathcal{F}$  must be visited infinitely often. The CTL formula under fairness constraints can be decided by a set of nested fixpoint computations:

$$\text{EG}_{\text{fair}} \text{true} = \nu Z. \text{EX} \bigwedge_{F_i \in \mathcal{F}} \text{E } Z \cup (Z \wedge F_i) ,$$

where  $\nu$  denotes the outer greatest fixpoint computation, and  $\text{EU}$  represents the embedded least fixpoint computations. When a monotonically increasing transform function  $f$  is applied repeatedly to a set  $Z$ , we define  $f^{(i)}(Z) = f(f(\dots f(Z)))$  and declare  $Z$  as a least fixpoint if  $f^{(i)}(Z) = f^{(i+1)}(Z)$ . Conversely, when a monotonically decreasing transform function  $g$  is applied repeatedly to a set  $Z$ , we define  $g^{(i)}(Z) = g(g(\dots g(Z)))$  and declare  $Z$  as a greatest fixpoint if  $g^{(i)}(Z) = g^{(i+1)}(Z)$ . When we evaluate the above formula through fixpoint computation, the initial value of the auxiliary iteration variable  $Z$  can be set to the entire universe.

For our running example,

$$F_0 = \{a0, b0, c0\}$$

$$Z^0 = \{a0, b0, c0, a1, b1, c1, a2, b2, c2, a3, b3, c3\}$$

$$\begin{aligned} Z^1 &= \text{EX } \text{E } Z^0 \cup (Z^0 \wedge F_0) \\ &= \text{EX } \text{E } \{a0, b0, c0, a1, b1, c1, a2, b2, c2, a3, b3, c3\} \cup \{a0, b0, c0\} \\ &= \text{EX } \{a1, b0\} \\ &= \{a1\} \end{aligned}$$

$$\begin{aligned} Z^2 &= \text{EX } \text{E } Z^1 \cup (Z^1 \wedge F_0) \\ &= \text{EX } \text{E } \{a1\} \cup \{ \} \\ &= \text{EX } \{ \} \\ &= \{ \} \end{aligned}$$

Since no state in the composed system satisfies  $\text{EG}_{\text{fair}} \text{ true}$ , the language is empty. This method for deciding  $\text{EG}_{\text{fair}} \text{ true}$  is known as the Emerson and Lei algorithm [EL86], which is the representative of a class of SCC hull algorithms [SRB02]. In general, the evaluation of EX and EU operators does not have to take the above alternating order; they can be computed in arbitrary orders without affecting the convergence of the fix-point [FFK<sup>+</sup>01, SRB02]. All SCC hull algorithms share the same worst-case complexity bound — they require  $O(\eta^2)$  symbolic steps, where  $\eta$  is the number of states of the composed model. A symbolic step is either a pre-image computation (finding predecessors through the evaluation of EX) or an image computation (finding successors through the evaluation of EY, the dual of EX).

Another way of checking language emptiness is to find all the strongly connected components (SCCs) and then check whether any of them satisfies all the acceptance conditions. If there exists a reachable non-trivial SCC that intersects every  $F_i \in \mathcal{F}$ , the language of the Büchi automaton is not empty. An SCC consisting of just one state without a self-loop is called *trivial*. In our running example, the reachable non-trivial SCCs of the composed system are  $\{a1\}$  and  $\{c1\}$ . Since none of the non-trivial SCCs intersects the fair set  $\{a0, b0, c0\}$ , the language of the system is empty.

An SCC is a maximal set of states such that there is a path between any two states. A reachable non-trivial SCC that intersects all acceptance conditions is called a fair SCC. An SCC that contains some initial states is called an initial SCC. An *SCC-closed set* of  $\mathcal{A}$  is the union of a collection of SCCs. The complete set of SCCs of  $\mathcal{A}$ , denoted by  $\Pi(\mathcal{A})$ , forms a partition of the states of  $\mathcal{A}$ . Likewise, the set of disjoint SCC-closed sets can also form a partition of the state space  $S$ . A SCC partition  $\Pi_1$  of  $S$  is a refinement of another partition  $\Pi_2$  of  $S$  if for every SCC or SCC closed set  $C_1 \in \Pi_1$ , there exists  $C_2 \in \Pi_2$  such that  $C_1 \subseteq C_2$ .

An *SCC (quotient) graph* is constructed from a graph by contracting each SCC into a node, merging parallel edges, and removing self-loops. The SCC graph of  $\mathcal{A}$ , denoted by  $\mathcal{Q}(\mathcal{A})$ , is a directed acyclic graph (DAG); it induces a partial order: the minimal (maximal) SCC has no incoming (outgoing) edge. Reachable fair SCCs, by definition, contain accepting runs that make the language non-empty. Therefore, a straightforward way of checking language emptiness is to compute all the reachable SCCs, and then check whether any of them is a fair SCC.

**OBSERVATION 2.8** *The language of a Büchi automaton is empty if and only if it does not have any reachable fair SCC.*

Tarjan’s explicit SCC algorithm using depth-first search [Tar72] can be used to decide language emptiness. The algorithm can be classified as an explicit-state algorithm because it traverses one state at a time. Tarjan’s algorithm has the best asymptotic complexity bound—linear in the number of states of the graph. However, for model checking industrial-scale systems, even the performance of such a linear time algorithm is not be good enough due the extremely large state space. A remedy to the search state explosion is a technique called “on-the-fly” model checking [GPVW95, Hol97], which avoids the construction of the entire state transition graph by visiting part of the state space at a time and constructing part of the graph as needed. Its fair cycle detection is based on two nested depth-first search procedures. Early termination, efficient hashing techniques, and partial order reduction can be used to reduce memory usage during the search and the number of interleavings that need to be inspected. The scalability issue in explicit-state enumeration makes them unsuitable for hardware designs, although they have been successful in verifying controllers and software.

Symbolic state space traversal techniques are another effective way of dealing with the extremely large state transition graphs. Instead of manipulating each individual state separately, symbolic algorithms manipulate sets of states. This is accomplished by representing the transition relation of the graph and sets of states as Boolean formulae, and conducting the search by directly manipulating the symbolic representations. By visiting a set of states at a time (as opposed to a single state), symbolic algorithms can traverse a very large state space using a reasonably small amount of time and memory. Thousands or even millions of states, for instance, can be visited in one symbolic step.

An example of symbolic graph algorithms in the context of LTL model checking is the SCC hull algorithms introduced earlier (of which the Emerson and Lei algorithm [EL86] is a representative), wherein each image or pre-image computation is considered as a symbolic step. There is also another class of SCC computation algorithms based on symbolic traversal, called the SCC enumeration algorithms [XB99, BRS99, GPP03]. Both SCC hull algorithms and SCC enumeration algorithms can be used for fair cycle detection and therefore can decide  $EG_{\text{fair}} \text{ true}$ ; however, some of the enumeration algorithms have better complexity bounds than SCC hull algorithms. For example, the Lockstep algorithm by Bloem *et al.* [BRS99] runs in  $O(\eta \log \eta)$  symbolic steps, and the more recent algorithm by Gentilini *et al.* [GPP03] runs in  $O(\eta)$  symbolic steps.



## 2.4 BDD-based Model Checking

In symbolic model checking, we manipulate sets of states instead of each individual state. Both the transition relation of the graph and the sets of states are represented by Boolean functions called *characteristic functions*, which are in turn represented by BDDs. Let the model be given in terms of

- a set of present-state variables  $x = \{x_1, \dots, x_m\}$ ,
- a set of next-state variables  $y = \{y_1, \dots, y_m\}$ ;
- a set of input variables  $w = \{w_1, \dots, w_n\}$ , and

the state transition graph can be represented symbolically by  $\langle T, I \rangle$ , where  $T(x, w, y)$  is the characteristic function of the transition relation, and  $I(x)$  is the characteristic function of the initial states. A *state* is a valuation of either the present-state or the next-state variables. For  $m$  state variables in the binary domain  $\mathbb{B} = \{0, 1\}$ , the total number of valuations is  $|\mathbb{B}|^m$ .

If a valuation of the present-state variables, denoted by  $\tilde{x}$ , makes the initial predicate  $I(\tilde{x})$  evaluate to **true**, the corresponding state is an initial state. Let  $\tilde{x}$ ,  $\tilde{y}$ , and  $\tilde{w}$  be the valuations of  $x$ ,  $y$ , and  $w$ , respectively; the transition relation  $T(\tilde{x}, \tilde{w}, \tilde{y})$  is **true** if and only if under the input condition  $\tilde{w}$ , there is a transition from the state  $\tilde{x}$  to the state  $\tilde{y}$ .

In our running example in Figure 2.2, the present-state variables, next-state variables, and inputs are  $\{x_1, x_0\}$ ,  $\{y_1, y_0\}$ , and  $w_0$ , respectively. The next-state functions of the two latches, in terms of the present-state variables and inputs, are:

$$\begin{aligned}\Delta_1 &= (x_1 \oplus x_0) \vee (x_1 \wedge x_0 \wedge \neg w_0) , \\ \Delta_0 &= (\neg x_1 \wedge \neg x_0 \wedge \wedge w_0) \vee (x_1 \wedge x_0 \wedge \neg w_0) .\end{aligned}$$

Note that  $\oplus$  denotes the exclusive OR operator. Boolean functions  $T$  and  $I$  are given as follows:

$$\begin{aligned}T &= (y_1 \leftrightarrow \Delta_1) \wedge (y_0 \leftrightarrow \Delta_0) , \\ I &= \neg x_1 \wedge \neg x_0 .\end{aligned}$$

When  $(x_1 = 0, x_0 = 0)$ ,  $(y_1 = 0, y_0 = 1)$ , and  $w_0 = 1$ , for instance, the transition relation  $T$  evaluates to **true**, meaning a valid transition exists from the state  $(0, 0)$  to the state  $(0, 1)$  under this input particular condition.

Computing the *image* or *pre-image* is the most fundamental step in symbolic model checking. The image of a set of states consists of all the

successors of these states in the state transition graph; the pre-image of a set of states consists of all their predecessors. In model checking, two existential CTL formulae,  $\text{EX}(D)$  and  $\text{EY}(D)$ , are used to represent the image and pre-image of the set of states  $D$  under the transition relation  $T$ . With a little abuse of notation, we are using  $\text{EX}$  and  $\text{EY}$  as both temporal operators as well as set operators. These two basic operations are defined as follows:

$$\begin{aligned}\text{EX}_T(D) &= \{s \mid \exists s' \in D : (s, s') \in T\} , \\ \text{EY}_T(D) &= \{s' \mid \exists s \in D : (s, s') \in T\} .\end{aligned}$$

When the context is clear, we will drop the subscripts and use  $\text{EX}$  and  $\text{EY}$  instead. Given the symbolic representation of the transition relation  $T$  and a set of states  $D$ , the image and pre-image of  $D$  are computed as follows:

$$\begin{aligned}\text{EX}_T(D) &= \exists y, w . T(x, w, y) \wedge D(y) , \\ \text{EY}_T(D) &= \exists x, w . T(x, w, y) \wedge D(x) .\end{aligned}$$

When we use them inside a fixpoint computation, we usually represent sets of states as Boolean functions in terms of the present-state variables only. Therefore, before pre-image computation and after image computation, we also need to simultaneously substitute the set of present-state variables with the corresponding next-state variables.

Many interesting temporal logic properties can be evaluated by applying  $\text{EX}$  and  $\text{EY}$  repeatedly, until a *fixpoint* is reached. The set of states that are reachable from  $I$ , for instance, can be computed by a least fixpoint computation as follows:

$$\text{EP } I = \mu Z . I \cup \text{EY}(Z) .$$

Here  $\text{EP } I$  denotes the set of reachable states and  $\mu$  represents the least fixpoint computation. In this computation, we have  $Z^0 = \emptyset$  and  $Z^{i+1} = I \cup \text{EY}(Z^i)$  for all  $i \geq 0$ . That is, we repeatedly compute the image of the set of already reached states starting from the initial states  $I$ , until the result stops growing. Similarly, the set of states from which  $D$  is reachable, denoted by  $\text{EF } D$ , can be computed by the least fixpoint computation

$$\text{EF } D = \mu Z . D \cup \text{EX}(Z) .$$

This fixpoint computation is often called the *backward reachability*.

The computation of  $\text{EG } D$ , on the other hand, corresponds to a greatest fixpoint computation. ( $\text{EG } p$  means that there is a path on which  $p$  always holds—in a finite state transition graph, such a path corresponds to a cycle.) It is defined as follows:

$$\text{EG } D = \nu Z . D \cap \text{EX}(Z) ,$$

where  $\nu$  represents the greatest fixpoint computation. In this computation, we have  $Z^0$  set to the entire universe and  $Z^{i+1} = D \cap \text{EX}(Z^i)$  for all  $i \geq 0$ .

The computation of  $\text{EG}_{\text{fair}} \text{true}$  also corresponds to a set of fixpoint computations. As pointed out in the previous section, this is a CTL property augmented with a set of Büchi fairness constraints, and it can be used to decide whether the language of a Büchi automaton is empty. The formula can be evaluated through fixpoint computations as follows:

$$\text{EG}_{\text{fair}} \text{true} = \nu Z. \text{EX} \bigwedge_{F_i \in \mathcal{F}} \text{E } Z \text{ U } (Z \wedge F_i) .$$

The evaluation corresponds to two nested fixpoint computations, a least fixpoint (EU) embedded in a greatest fixpoint ( $\nu Z. \text{EX}$ ). In the previous section, we have given a small example to illustrate the evaluation of this formula.

As mentioned before, we can use symbolic algorithms to enumerate the SCCs in a graph [XB99, BRS99, GPP03]. Conceptually, an SCC enumeration algorithm works as follows (here we take the algorithm in [XB99] as an example, for it is the simplest among the three and it serves as a stepping stone for understanding the other two). First, we pick an arbitrary state  $v$  as *seed* and compute both  $\text{EF } v$  and  $\text{EP } v$ .  $\text{EF } v$  consists of states that can reach  $v$  and  $\text{EP } v$  consists of states reachable from  $v$ . We then intersect the two sets of states to get an SCC (and the intersection is guaranteed to be an SCC). If the SCC does not intersect with all the fair sets  $F_i \in \mathcal{F}$ , we remove it from the graph and pick another seed from the remaining graph. We keep doing that until we found an SCC satisfying all the acceptance conditions, or no state is left in the graph. Although SCC enumeration algorithms may have better complexity bounds than SCC hull algorithms, for industrial-scale systems, applying any of these algorithms directly to the concrete model remains prohibitively expensive.

For certain subclasses of LTL properties, there exist specialized algorithms that often are more efficient than the evaluation of  $\text{EG}_{\text{fair}} \text{true}$ . One way of finding these subclasses is to classify LTL properties by the strength of the corresponding Büchi automata. According to [BRS99], the strength of a property Büchi automaton can be classified as *strong*, *weak*, and *terminal*. If the property automaton is classified as *strong*, checking the language emptiness of the composed system requires the evaluation of the general formula  $\text{EG}_{\text{fair}} \text{true}$ . Whenever the property automaton is *weak* or *terminal*, language emptiness checking in the composed system only requires the evaluation of  $\text{EF fair}$  or  $\text{EF EG fair}$ , respectively. Note that the latter two formulae are much easier to evaluate

since they corresponds to a single fixpoint computation or two fixpoint computations aligned in a row, rather than two fixpoint computations but with one nested inside the other as in  $\text{EG}_{\text{fair}} \text{true}$ .

Let us take the invariant property  $\text{G}p$  as an example, whose corresponding property automaton is denoted by  $\mathcal{A}_{\text{F}\neg p}$ . The property automaton is given at the left-hand side of Figure 2.6. The only acceptance condition of the automaton is  $\{2\}$ , or  $\mathcal{F} = \{\{2\}\}$ . The SCC  $\{2\}$  is a fair SCC since it satisfies the acceptance condition; furthermore, the SCC is maximal in the sense that no outgoing transition exists. For the automaton at the left-hand side, we can mark State 1 accepting as well; that is,  $\text{fair} = \{1, 2\}$ . The automaton remains equivalent to the original one because both accept the same  $\omega$ -regular language. However, this new property automaton can be classified as a *terminal* automaton although according to the definition in [BRS99], the original one is classified as weak. For a terminal property automaton, the language of the composed system is empty as long as no state in the fair SCC is reachable. This is equivalent to evaluating the much simpler formula  $\text{EF fair}$  (which has a similar complexity as the reachability analysis). Note also that when  $\text{EF fair}$  is used instead of  $\text{EG}_{\text{fair}} \text{true}$ , we may end up producing a shorter counterexample.

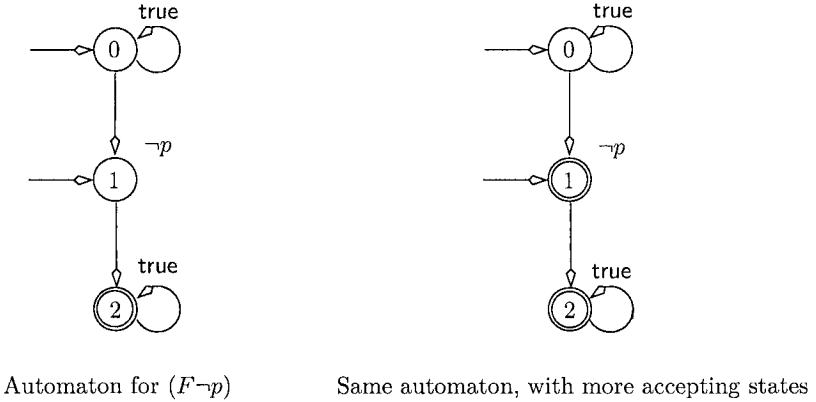


Figure 2.6. Two terminal generalized Büchi automata.

### 2.4.1 Binary Decision Diagrams

Set operations encountered in symbolic fixpoint computations, including *intersection*, *union*, and *existential quantification*, are implemented as BDD operations. BDDs are an efficient data structure for repre-

senting Boolean functions. BDD in its current form is both *reduced* and *ordered*, called the Reduced Ordered BDD (RO-BDD). ROBDD was first introduced by Bryant [Bry86], although the general ideas of branching programs have been available for a long time in the theoretical computer science literature. Symbolic model checking based on BDDs, introduced by McMillan [McM94], is considered as a major breakthrough in increasing the model checker's capacity, leading to the subsequently widespread acceptance of model checking in the computer hardware industry.

Given a Boolean function, we can build a binary decision tree by obeying a linear order of decision variables; that is, along any path from root to leaf, the variables appear in the same order and no variable appears more than once. We further restrict the form of the decision tree by repeatedly merging any duplicate nodes and removing nodes whose *if* and *else* branches are pointing to the same child node. The resulting data structure is a directed acyclic graph. Conceptually, this is how we construct the ROBDD for a given Boolean function. In practice, BDDs are created directly in the fully reduced form without the need to build the original decision tree in the first place. BDDs representing multiple Boolean functions are also merged into a single directed graph to increase the sharing; such a graph would have multiple roots, one for each Boolean function.

The formal definition of a BDD is given as follows:

**DEFINITION 2.9** *A BDD is a directed acyclic graph  $(\Phi \cup V \cup \{1\}, E)$  representing a set of functions  $f_i : \{0, 1\}^n \rightarrow \{0, 1\}$ . The nodes  $\Phi \cup V \cup \{1\}$  are partitioned into three subsets.*

- *1 is the only terminal node whose out-degree is 0.*
- *$V$  is the set of internal nodes whose out-degree is 2 and whose in-degree is 1. Every node  $v \in V$  corresponds to a Boolean variable  $l(v)$  in the support of functions  $\{f_i\}$ ; the  $n$  variables  $\{l(v)\}$  in the entire graph are ordered as follows: if  $v_j$  is a descendant of  $v_i$ , then  $l(v_j) < l(v_i)$ .*
- *$\Phi$  is the set of function nodes whose out-degree is 1 and whose in-degree is 0; the function nodes are in one-to-one correspondence with the  $f_i$ 's.*

*$E$  is the set of edges connecting the nodes. The outgoing edge of function nodes may have the complement attribute. The two outgoing edges for a node  $v \in V$  are labeled  $T$  and  $E$ , respectively. The  $E$  edges may have the complement attribute. We write  $(l(v), T(v), E(v))$  to indicate an internal node and its two outgoing edges.*

The set of functions represented by a BDD is defined as follows: (1) The function of the only terminal node, **1**, is **true**. (2) The function of a regular edge is the function of the head node; the function of a *complement* edge is the complement of the function of the head node. (3) The function of a node  $v \in V$  is  $l(v) \wedge f_T \vee \neg l(v) \wedge f_E$ , where  $f_T$  and  $f_E$  are the functions of its T and E edges. (4) The function of  $\phi \in \Phi$  is the function of its outgoing edge.

BDD provides a very compact representation for many Boolean functions found in practice, although in the worst case the size of a BDD may become exponential with respect to the number of support variables. (An example for the worst-case blowup is a multiplier, which is known to have an exponential number of BDD nodes regardless of the BDD variable order [Bry86].) In addition to the compactness, BDDs are also easy to manipulate. Efficient algorithms exist for almost all the common set-theoretic operations. For example, the intersection or union of two BDDs takes time proportional to the product of their respective sizes in the worst case.

BDDs are also a canonical representation in the sense that with a fixed variable ordering, every Boolean function has a unique BDD representation. Therefore, checking whether two Boolean functions are the same is reduced to a pointer comparison. Given a BDD, complementation or the validity check takes constant time as well.

The complexity of symbolic model checking depends on the size of the BDDs involved in the symbolic steps, such as the BDDs that represent the transition relation and sets of states. Because of this, the search for heuristics to avoid the BDD blow-up in the context of image computation and symbolic fixpoint computation has been one of the major research topics in formal verification.

CU Decision Diagram (CUDD) [Som] is a public-domain decision diagram package developed in the University of Colorado. CUDD has been used widely in both industry and academia. The package provides a large set of operations to manipulate BDDs, Algebraic Decision Diagrams (ADDs) [BFG<sup>+</sup>93], and Zero-suppressed Binary Decision Diagrams (ZDDs) [Min93]. The latter two are variants of BDDs. In particular, ADDs represent function from  $\mathbb{B}^m$  to an arbitrary set (e.g., the integer domain), as opposed to  $\mathbb{B}$  in BDDs. ZDDs represent switching functions like BDDs, but they are much more efficient than BDDs when the functions to be represented are characteristic functions of cube sets, or in general, when the ON-set of the function to be represented is very sparse. They are inferior to BDDs in other cases. The CUDD package also provides functions to convert BDDs into ADDs or ZDDs and vice versa, and a large assortment of variable reordering methods.

## 2.5 SAT and Bounded Model Checking

SAT based Bounded Model Checking (BMC) is a complementary technique to BDD based symbolic model checking. BMC was first introduced by Biere *et al.* in [BCCZ99]. Given a model and an LTL property, BMC searches in the given model for counterexamples of a finite length. The existence of a finite length counterexample is encoded as a Boolean formula, which is satisfiable if and only if the counterexample exists. The satisfiability problem of a Boolean formula can be decided by a SAT solver. Since modern SAT solvers [SS96, MMZ<sup>+</sup>01, GN02] often suffer less from the potential search space explosion, in practice, SAT based bounded model checking can handle some industrial-scale circuits that are beyond the reach of BDD based techniques.

In bounded model checking, one can keep increasing the counterexample length  $k$  (also called the unrolling depth) until either a counterexample is found or  $k$  exceeds a predetermined *completeness threshold*. A completeness threshold is a constant  $k_c$  such that if we cannot find any counterexample shorter than or equal to  $k_c$ , we have proved that the property holds. It is clear that  $k_c \leq \eta$ , where  $\eta$  is the number of states of the Kripke structure, since any finite run of the Kripke structure with distinct states cannot be longer than that. Therefore, BMC can be regarded as transforming the PSPACE-complete problem of LTL model checking into a finite number of Boolean satisfiability checks. Although each individual SAT problem in this process is NP-complete, the total number of SAT problems, or  $k_c$ , can be exponential with respect to the number of state variables. In practice, the SAT checks often slow down significantly after  $k$  goes beyond a few hundred steps.

A better completeness threshold than  $\eta$  would be the diameter of the state transition graph [KS03], i.e., the length of the longest shortest path between any two states. For safety properties, one can go one step further and use the reachable diameter of the graph, which is the length of the longest shortest path between an initial state and another state. While the diameter of a design may be exponential in the number of its state elements, Baumgartner and Kuehlmann [BK04] have observed that in practice it often ranges from tens to a few hundred regardless of design size. They also proposed a general approach for enabling the use of structural transformations to tighten the bounds obtained by arbitrary diameter approximation techniques. However, despite these previous research works, computing a tight bound of the diameter of an extremely large graph remains very hard in practice. In the absence of a reasonably small completeness threshold, people use BMC primarily for detecting bugs rather than for proving properties.

Other SAT based methods have also been used to prove properties. This includes methods based on induction [SSS00, dRS03, GGW<sup>+</sup>03a], unbounded model checking through the enumeration of all SAT solutions [GYAG00, McM02, KP03, GGA04], and the interpolation based method [MA03]. Other more recent works have extended the BMC induction proof methods to handle liveness properties [AS04, GGA05a].

The Boolean formula used to encode the existence of a finite length counterexample consists of two subformulae:  $\Phi = \Phi_M \wedge \Phi_P$ . The first subformula, denoted by  $\Phi_M$ , captures all the length  $k$  execution traces that are possible in the Kripke structure, all of which start from the initial states. The second subformula, denoted by  $\Phi_P$ , captures the constraint for a length  $k$  path to violate the given property. The conjunction of the above two subformulae captures all the length  $k$  counterexamples with respect to the property. Such a counterexample exists if and only if the Boolean formula has a satisfiable assignment.

First, we explain how to create the subformula  $\Phi_M$ . We use  $V$  to represent the set of state variables (or latches) and  $U$  to represent the rest of the signals (primary inputs, outputs, and signals of internal logic gates). We then replicate these variables at every clock cycle: we use  $V^i = \{v_1^i, \dots, v_n^i\}$  to represent the set of state variables at the  $i$ -th time frame and  $U^i = \{u_1^i, \dots, u_m^i\}$  to represent the set of other signals at the  $i$ -th time frame. Now we can unroll the sequential circuit by making multiple copies of the symbolic transition relation, and create a combination circuit. When the BMC unrolling depth is  $k$ ,

$$\Phi_M = I(V^0) \wedge \bigwedge_{1 \leq i \leq k} T(V^{i-1}, U^{i-1}, V^i) ,$$

where  $I(V^0)$  requires that all paths must start from an initial state, and the rest is the conjunction of  $k$  copies of the transition relation.

The transition relation copy at the  $i$ -th time frame is denoted by  $T(V^{i-1}, U^{i-1}, V^i)$ , which is the conjunction of elementary transition relations,

$$T(V^{i-1}, U^{i-1}, V^i) = \bigwedge_{1 \leq j \leq n} (v_j^i \leftrightarrow u_j^{i-1}) \wedge \bigwedge_{1 \leq j \leq m} T_j(U^{i-1}, V^i) .$$

Each  $T_j$  is called a *gate relation* for it describes the behavior of a logic gate. For instance, if  $u_j$  is the output variable of a two-input *and* gate with inputs  $u_l$  and  $u_r$ , then  $T_j = u_j \leftrightarrow (u_l \wedge u_r)$ . However, if  $u_j$  is a primary input to the circuit, then  $T_j = 1$ . Each term of the form  $(v_j^i \leftrightarrow u_l^{i-1})$  equates a next-state variable to a combinational variable, describing that the output of a logic gate is fed to the data input of the  $j$ -



th register. Adjacent transition relation copies are effectively connected together through the use of shared state variables in  $V^i$ .

Next we explain the creation of the subformula  $\Phi_P$ , which states that a path of length  $k$  must violate the given property. To explain the basic idea, we use the invariant property  $Gp$  as an example. For the encoding of a general LTL formula in BMC, the readers are referred to the original BMC paper [BCCZ99]. Note that the property  $Gp$  fails if and only if there is a path of length  $k$  from an initial state to a state labeled  $\neg p$ . Therefore,

$$\Phi_P = \neg P(V^k) ,$$

which indicates that the last state is a “bad” state. We use  $P(V^k)$  to denote the predicate that the state  $V^k$  satisfies the propositional formula  $p$ . In general  $\Phi_P$  should be the conjunction of  $\neg P(V^i)$  for  $0 \leq i \leq k$ . However, if we start BMC with  $k = 0$  and keep increasing the unrolling depth by 1 at a time, by the time it reaches  $k$  we know that the “bad” state cannot be found in the first  $(k - 1)$  depths; therefore,  $\Phi_P$  can be simplified into  $\neg P(V^k)$ . To summarize, the entire BMC instance at the unrolling depth  $k$  is given as follows,

$$\Phi = I(V^0) \wedge \bigwedge_{1 \leq i \leq k} T(V^{i-1}, U^{i-1}, V^i) \wedge \neg P(V^k) .$$

This formula can be viewed as a pure combinational circuit with some environmental constraints. Figure 2.7 illustrates such a view for the unrolling depth 2.

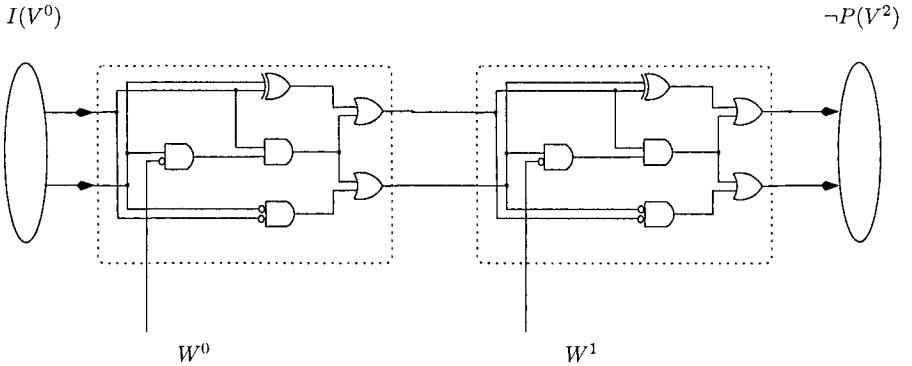


Figure 2.7. A bounded model checking instance.

Now, we explain how to convert the Boolean formula  $\Phi$  into the Conjunctive Normal Form (CNF). This step is necessary because in practice,

we often use an off-the-shelf Boolean SAT solver to decide  $\Phi$ , and most of the modern SAT solvers accept the CNF input format. Furthermore, CNF is also adopted by many solvers as an efficient data structure for internal representation. A CNF formula is the conjunction of a set of *clauses*, each of which is a disjunction of *literals*. A *literal* is the positive (or negative) phase of a Boolean *variable*. As an example, the following formula fragment

$$(a \vee \neg c) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee c) \wedge \dots$$

has three variables ( $a$ ,  $b$ , and  $c$ ), six literals ( $a$ ,  $\neg a$ ,  $b$ ,  $\neg b$ ,  $c$ , and  $\neg c$ ), and three clauses.

Both general Boolean formulae and combinational circuits can be converted into CNF formulae in linear time if we are allowed to add auxiliary variables. The result CNF formula is also linear with respect to the size of the input formula or the size of the input circuit. Since the transition relation of a Kripke structure can be represented as a network of logic gates, we only need to consider the problem of encoding the individual combinational logic gates as conjunctions of clauses. The solution to the latter problem is actually very straightforward [CLR90]. For instance, a two-input *and* gate  $u_j$  with inputs  $u_l$  and  $u_r$  has the following set of clauses:

$$(u_l \vee \neg u_j) \wedge (u_r \vee \neg u_j) \wedge (\neg u_l \vee \neg u_r \vee u_j) .$$

Finally, we review a procedure used in many modern SAT solvers [SS96, MMZ<sup>+</sup>01, GN02] to decide the satisfiability of a CNF formula. A formula is *satisfiable* if and only if there exists a set of assignments to the variables that makes the formula **true**. It is clear that for the entire CNF formula to be satisfiable, each individual clause must also be satisfiable. A clause is satisfiable if and only if at least one of its literals evaluates to **true**. The SAT problem of a CNF formula can be solved by the Davis-Longeman-Loveland (DLL) recursive search procedure [DLL62]. Its basic steps are making decisions and propagating the implications of these decisions. Selecting a literal and making it **true** is called a *decision*. If a clause has only one unassigned literal and all the other literals are **false**, it is called a unit clause. Every unit clause triggers an implication—its only unassigned literal has to be **true**, otherwise, the clause is no longer satisfiable. The process of applying implications iteratively until no unit clause is left is called *Boolean Constraint Propagation (BCP)*.

A decision and the corresponding BCP restrict our attention into a subformula or a subset of the original clauses, since the rest of the clauses have been made **true**. If we keep making decisions on free variables and performing BCP until no subformula remains to be decided, the formula

is proved to be satisfiable. However, if the remaining subformula is not satisfiable (i.e., some of its clauses become false), we need to backtrack and flip some of the previous assignments we made earlier.

```

SATCHECK( )
{
    while (true)
    {
        if ( MAKEDECISION( ) )
        {
            while ( BCP( ) == CONFLICT )
            {
                level = CONFLICTANALYSIS( );
                if (level < 0)
                    return UNSAT;
                else
                    BACKTRACK(level);
            }
        }
        else
            return SAT;
    }
}

```

Figure 2.8. The DLL Boolean SAT procedure.

The pseudo code of a DLL procedure is given in Figure 2.8. It makes decisions and then applies BCP inside the *while* loop. If all the variables have been assigned and no conflict occurs, the procedure MAKEDECISION will return false and the procedure SATCHECK will return a complete set of variable assignments. If a conflict occurs after a partial set of assignments—some clauses become false, the procedure BCP will return CONFLICT, indicating that a previous decision is not appropriate. Inside the procedure CONFLICTANALYSIS, the level of that previous decision is identified by conflict analysis [SS96], following which, the inappropriate decision is recovered by backtracking. Before backtracking, a conflict clause learned from this analysis can be added to the clause database (i.e., conjoined with the original formula) to prevent the search from repeating this mistake in the future. When the backtracking level is less than 0, the given formula is proved to be unsatisfiable—there is a conflict before we make any decision.

There are also efficient implementations of the Boolean SAT solver that do not rely solely on the CNF representation [KGP01, KPKG02]. Since many SAT problems in EDA, including bounded model check-

ing, are typically derived from the circuit structure, it is advantageous to work directly on the circuit representation, and use circuit-specific heuristics to guide the search. In [GAG<sup>+</sup>02], Ganai *et al.* proposed a hybrid SAT solver which combines the benefits of circuit-based and CNF SAT techniques. Other more recent efforts have also combined the advantages of multiple symbolic representations, including circuit graphs, CNF, and BDDs [GGW<sup>+</sup>03b, IPC03, LWCH03, JS04, JAS04]

## 2.6 Abstraction Refinement Framework

Abstraction refinement was first introduced by Kurshan [Kur94] in checking linear time properties specified by  $\omega$ -regular automata. It is an important technique to bridge the capacity gap between the model checker and large digital systems. When a model cannot be directly handled by the model checker due to the limited capacity of the model checking algorithms, abstraction can be used to simplify the model by removing the information that is irrelevant to verification. To simplify verification, we want to retain only the relevant details with respect to deciding the property at hand. The key issue in abstraction refinement is identifying in advance which part of the model is relevant and which is not.

There are automatic techniques for computing a simplified model in which an certain class of temporal logic properties can be preserved. For instance, bi-simulation based reduction [Mil71, DHWT91] preserves the full propositional  $\mu$ -calculus (hence the entire CTL since all CTL formulae can be evaluated through the translation to fixpoint computations in propositional  $\mu$ -calculus). A nice feature of these techniques is that we only need to compute the reduction for a given model once, and then use the simplified model to check all kinds of properties in that class. In practice, however, bi-simulation and other property preserving abstractions are less attractive because they are either hard to compute, or do not achieve a drastic reduction. A previous study by Fisler and Vardi [FV99] has demonstrated that bi-simulation relation is often expensive to compute and bi-simulation based simplification does not speed up CTL model checking.

A more practical abstraction approach is called property driven abstraction, which often results in a significantly smaller model that preserves or partially preserves a given property (as opposed to a class of properties). This abstraction approach is frequently used in the iterative abstraction refinement loop. There are various ways of deriving such an abstract model [BSV93, Lon93, CHM<sup>+</sup>96a]. Most of them create abstract models that are upper bounds or over-approximations of the exact system, which may have more behavior than the concrete model.

Over-approximated abstraction may produce *false negatives* when being used to verify universal properties such as  $\text{AG } p$ : If a property holds in the abstract model, it also holds in the concrete model; however, if the property fails in the abstract model, it may still pass in the concrete model—in this case, the property is still undecided.

There are also lower bounds or under-approximations of the exact system. These abstractions are conservative as well because they may produce *false positives* when being used to check universal properties. Since the abstract models have less behavior than the exact system, if a counterexample is found in the abstract model, then it is also a counterexample in the exact system. However, if no counterexample exists in the abstract model, we cannot conclude that the property is true. In other words, these abstractions can only refute a property but cannot prove it. Note that one can also use under-approximations and over-approximations simultaneously in a single iterative refinement process, to tighten up abstraction from both ends.

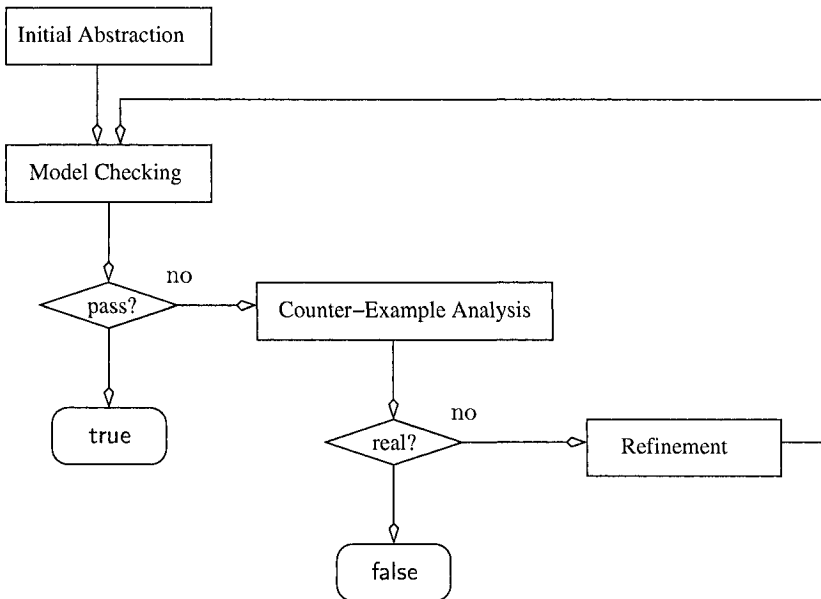


Figure 2.9. The abstraction refinement framework.

Since the property driven abstraction is conservative, we need an iterative process to refine the abstract model until it becomes deciding. In abstraction refinement, one seeks the simplest abstraction that can

either prove or refute the given property. Such abstraction is called the *final* or *deciding* abstraction for the given model checking problem. Figure 2.9 shows the generic framework for iterative abstraction refinement. Given a model and an LTL property, for instance, one can build a very coarse initial abstraction that is an over-approximation of the concrete model [Kur94, HHK96]. We then use a model checker to decide whether the abstract model satisfies the property. If the property is satisfied by the abstract model, it is also satisfied by the concrete model. If the property fails in the abstract model, there is no conclusive result yet.

At this point, the model checker returns an abstract counterexample showing how the property is violated. Inside counterexample analysis, we check whether this abstract counterexample contains a valid path in the concrete model. One way of doing that is using the *concretization test* to reconstruct the abstract path in the concrete model [CGJ<sup>+</sup>00, WHL<sup>+</sup>01]. If this is possible and we find a real path within the given abstract counterexample, the property is refuted. Otherwise, the abstract counterexample is declared as *spurious*, which means that some important information of the exact system is missing and therefore the current abstraction needs to be refined.

During refinement, one can use spurious counterexamples to guide the identification of missing information in the current abstract model [Kur94, CGJ<sup>+</sup>00]. Often, the immediate goal in counterexample guided refinement is to remove the set of spurious counterexamples. That is, one searches for a set of refinement variables such that, adding them into the current abstract model removes the spurious counterexamples.

After computing the set of refinement variables, we can build the new abstract model by including their corresponding bit transition relations. We then start the model checker again. This iterative process terminates when either the property is decided, or the available computing resources (i.e., CPU time and memory) are depleted.

Abstraction Refinement for Large Scale Model Checking

Wang, C.; Hachtel, G.D.; Somenzi, F.

2006, XIV, 179 p., Hardcover

ISBN: 978-0-387-34155-2