Model Checking

Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled

The MIT Press
Cambridge, Massachusetts
London, England

1 Introduction

Model checking is an automatic technique for verifying finite state concurrent systems. It has a number of advantages over traditional approaches to this problem that are based on simulation, testing, and deductive reasoning. The method has been used successfully in practice to verify complex sequential circuit designs and communication protocols. The main challenge in model checking is dealing with the *state space explosion* problem. This problem occurs in systems with many components that can interact with each other or systems that have data structures that can assume many different values (for example, the data path of a circuit). In such cases the number of global states can be enormous. During the past ten years considerable progress has been made in dealing with this problem. In this chapter we compare model checking with other formal methods for verifying hardware and software designs. We describe how model checking is used to verify complex system designs. We also trace the development of different model checking algorithms and discuss various approaches that have been proposed for dealing with the state explosion problem.

1.1 The Need for Formal Methods

Today, hardware and software systems are widely used in applications where failure is unacceptable: electronic commerce, telephone switching networks, highway and air traffic control systems, medical instruments, and other examples too numerous to list. We frequently read of incidents where some failure is caused by an error in a hardware or software system. A recent example of such a failure is the Ariane 5 rocket, which exploded on June 4, 1996, less than forty seconds after it was launched. The committee that investigated the accident found that it was caused by a software error in the computer that was responsible for calculating the rocket's movement. During the launch, an exception occurred when a large 64-bit floating point number was converted to a 16-bit signed integer. This conversion was not protected by code for handling exceptions and caused the computer to fail. The same error also caused the backup computer to fail. As a result incorrect attitude data was transmitted to the on-board computer, which caused the destruction of the rocket. The team investigating the failure suggested that several measures be taken in order to prevent similar incidents in the future, including the verification of the Ariane 5 software.

Clearly, the need for reliable hardware and software systems is critical. As the involvement of such systems in our lives increases, so too does the burden for ensuring their correctness. Unfortunately, it is no longer feasible to shut down a malfunctioning system in order to restore safety. We are very much dependent on such systems for continuous

operation; in fact, in some cases, devices are less safe when they are shut down. Even when failure is not life-threatening, the consequences of having to replace critical code or circuitry can be economically devastating.

Because of the success of the *Internet* and *embedded systems* in automobiles, airplanes, and other safety critical systems, we are likely to become even more dependent on the proper functioning of computing devices in the future. In fact, the pace of change will likely accelerate in coming years. Because of this rapid growth in technology, it will become even more important to develop methods that increase our confidence in the correctness of such systems.

1.2 Hardware and Software Verification

The principal validation methods for complex systems are simulation, testing, deductive verification, and model checking. Simulation and testing [202] both involve making experiments before deploying the system in the field. While simulation is performed on an abstraction or a model of the system, testing is performed on the actual product. In the case of circuits, simulation is performed on the design of the circuit, whereas testing is performed on the circuit itself. In both cases, these methods typically inject signals at certain points in the system and observe the resulting signals at other points. For software, simulation and testing usually involve providing certain inputs and observing the corresponding outputs. These methods can be a cost-efficient way to find many errors. However, checking *all* of the possible interactions and potential pitfalls using simulation and testing techniques is rarely possible.

The term *deductive verification* normally refers to the use of axioms and proof rules to prove the correctness of systems. In early research on deductive verification, the main focus was on guaranteeing the correctness of critical systems. It was assumed that the importance of their correct behavior was so great that the developer or a verification expert (usually a mathematician or a logician) would spend whatever time was required for verifying the system. Initially, such proofs were constructed entirely by hand. Eventually, researchers realized that software tools could be developed to enforce the correct use of axioms and proof rules. Such tools can also apply a systematic search to suggest various ways to progress from the current stage of the proof.

The importance of deductive verification is widely recognized by computer scientists. It has significantly influenced the area of software development (for example, the notion of an *invariant* originated in research on deductive verification). However, deductive verification is a time-consuming process that can be performed only by experts

who are educated in logical reasoning and have considerable experience. The proof of a single protocol or circuit can last days or months. Consequently, use of deductive verification is rare. It is applied primarily to highly sensitive systems such as *security protocols*, where enough resources need to be invested to guarantee their safe usage.

It is important to realize that some mathematical tasks cannot be performed by an algorithm. The theory of *computability* [142] provides limitations on what can be decided by an algorithm. In particular, it shows that there cannot be an algorithm that decides whether an arbitrary computer program (written in some programming language like C or Pascal) terminates. This immediately limits what can be verified automatically. In particular, correct termination of programs cannot be verified automatically in general. Thus, most proof systems cannot be completely automated.

An advantage of deductive verification is that it can be used for reasoning about infinite state systems. This task can be automated to a limited extent. However, even if the property to be verified is true, no limit can be placed on the amount of time or memory that may be needed in order to find a proof.

Model checking is a technique for verifying finite state concurrent systems. One benefit of this restriction is that verification can be performed automatically. The procedure normally uses an exhaustive search of the state space of the system to determine if some specification is true or not. Given sufficient resources, the procedure will always *terminate* with a **yes/no** answer. Moreover, it can be implemented by algorithms with reasonable efficiency, which can be run on moderate-sized machines (but usually not on an average desktop computer).

Although the restriction to finite state systems may seem to be a major disadvantage, model checking is applicable to several very important classes of systems. Hardware controllers are finite state systems, and so are many communication protocols. In some cases, systems that are not finite state may be verified using model checking in combination with various abstraction and induction principles. Finally, in many cases errors can be found by restricting unbounded data structures to specific instances that are finite state. For example, programs with unbounded message queues can be debugged by restricting the size of the queues to a small number like two or three.

Because model-checking can be performed automatically, it is preferable to deductive verification, whenever it can be applied. However, there will always be some critical applications in which theorem proving is necessary for complete verification. An exciting new research direction [220] attempts to integrate deductive verification and model checking, so that the finite state parts of a complex system can be verified automatically.

1.3 The Process of Model Checking

Applying model checking to a design consists of several tasks, each of which will be discussed in detail later in this book.

Modeling The first task is to convert a design into a formalism accepted by a model checking tool. In many cases, this is simply a compilation task. In other cases, owing to limitations on time and memory, the modeling of a design may require the use of abstraction to eliminate irrelevant or unimportant details.

Specification Before verification, it is necessary to state the properties that the design must satisfy. The specification is usually given in some logical formalism. For hardware and software systems, it is common to use *temporal logic*, which can assert how the behavior of the system evolves over time.

An important issue in specification is *completeness*. Model checking provides means for checking that a model of the design satisfies a given specification, but it is impossible to determine whether the given specification covers all the properties that the system should satisfy.

Verification Ideally the verification is completely automatic. However, in practice it often involves human assistance. One such manual activity is the analysis of the verification results. In case of a negative result, the user is often provided with an error trace. This can be used as a counterexample for the checked property and can help the designer in tracking down where the error occurred. In this case, analyzing the error trace may require a modification to the system and reapplication of the model checking algorithm.

An error trace can also result from incorrect modeling of the system or from an incorrect specification (often called a *false negative*). The error trace can also be useful in identifying and fixing these two problems. A final possibility is that the verification task will fail to terminate normally, due to the size of the model, which is too large to fit into the computer memory. In this case, it may be necessary to redo the verification after changing some of the parameters of the model checker or by adjusting the model (e.g., by using additional abstractions).

1.4 Temporal Logic and Model Checking

Temporal logics have proved to be useful for specifying concurrent systems, because they can describe the ordering of events in time without introducing time explicitly. They were originally developed by philosophers for investigating the way that time is used in natural language arguments [145]. Although a number of different temporal logics have been

studied, most have an operator like $\mathbf{G}\ f$ that is true in the present if f is always true in the future (*i.e.*, if f is globally true). To assert that two events e_1 and e_2 never occur at the same time, one would write $\mathbf{G}(\neg e_1 \vee \neg e_2)$. Temporal logics are often classified according to whether time is assumed to have a *linear* or a *branching* structure. In this book the meaning of a temporal logic formula will always be determined with respect to a labeled state-transition graph; for historical reasons such structures are called *Kripke structures* [145].

Several researchers, including Burstall [48], Kröger [158] and Pnueli [216], have proposed using temporal logic for reasoning about computer programs. However, Pnueli [216] was the first to use temporal logic for reasoning about concurrency. His approach involved proving properties of the program under consideration from a set of axioms that described the behavior of the individual statements in the program. The method was extended to sequential circuits by Bochmann [25] and Malachi and Owicki [184]. Since proofs were constructed by hand, the technique was often difficult to use in practice.

The introduction of temporal-logic model checking algorithms by Clarke and Emerson [61, 103] in the early 1980s allowed this type of reasoning to be automated. Because checking that a single model satisfies a formula is much easier than proving the validity of a formula for all models, it was possible to implement this technique very efficiently. The algorithm developed by Clarke and Emerson for the branching-time logic CTL was polynomial in both the size of the model determined by the program under consideration and in the length of its specification in temporal logic. They also showed how *fairness* [120] could be handled without changing the complexity of the algorithm. This was an important step in that the correctness of many concurrent programs depends on some type of fairness assumption; for example, absence of starvation in a mutual exclusion algorithm may depend on the assumption that each process makes progress infinitely often.

At roughly the same time Quielle and Sifakis [219] gave a model checking algorithm for a subset of CTL, but they did not analyze its complexity. Later Clarke, Emerson, and Sistla [63] devised an improved algorithm that was linear in the product of the length of the formula and the size of the state transition graph. The algorithm was implemented in the EMC model checker, which was widely distributed and used to check a number of network protocols and sequential circuits [28, 29, 30, 31, 63, 98, 197]. Early model checking systems were able to check state transition graphs with between 10^4 and 10^5 states at a rate of about 100 states per second for typical formulas. In spite of these limitations, model checking systems were used successfully to find previously unknown errors in several published circuit designs.

Sistla and Clarke [232, 233] analyzed the model checking problem for a variety of temporal logics and showed, in particular, that for linear temporal logic (LTL) the problem was PSPACE-complete. Pnueli and Lichtenstein [173] reanalyzed the complexity of checking

linear-time formulas and discovered that although the complexity appears exponential in the length of the formula, it is linear in the size of the global state graph. Based on this observation, they argued that the high complexity of linear-time model checking might still be acceptable for short formulas. The same year, Fujita [119] implemented a tableau based verification system for LTL formulas and showed how it could be used for hardware verification.

CTL* is a very expressive logic that combines both branching-time and linear-time operators. The model checking problem for this logic was first considered in a paper by Clarke, Emerson, and Sistla [62], where it was shown to be PSPACE-complete, establishing that it is in the same general complexity class as the model checking problem for LTL. This result can be sharpened to show that CTL* and LTL model checking are of the same algorithmic complexity (up to a constant factor) in both the size of the state graph and the size of the formula. Thus, for purposes of model checking, there is no practical complexity advantage to restricting oneself to a linear temporal logic [106].

Alternative techniques for verifying concurrent systems have been proposed by a number of other researchers. Many of these approaches use automata for specifications as well as for implementations. The implementation is checked to see whether its behavior conforms to that of the specification. Because the same type of model is used for both implementation and specification, an implementation at one level can also be used as a specification for the next level of refinement. The use of language containment is implicit in the work of Kurshan [1], which ultimately resulted in the development of a powerful verifier called COSPAN [132, 133, 162]. Vardi and Wolper [245] first proposed the use of ω-automata (automata over infinite words) for automated verification. They showed how the linear temporal logic model checking problem could be formulated in terms of language containment between ω-automata. Other notions of conformance between the automata have also been considered, including observational equivalence [77, 196, 224], and various refinement relations [77, 195, 223].

1.5 Symbolic Algorithms

In the original implementation of the model checking algorithm, transition relations were represented explicitly by adjacency lists. For concurrent systems with small numbers of processes, the number of states was usually fairly small, and the approach was often quite practical. In systems with many concurrent parts however, the number of states in the global state transition graph was too large to handle. In the fall of 1987, McMillan [46, 191], then a graduate student at Carnegie Mellon University, realized that by using a symbolic representation for the state transition graphs, much larger systems could be

verified. The new symbolic representation was based on Bryant's *ordered binary decision diagrams* (OBDDs) [34]. OBDDs provide a canonical form for boolean formulas that is often substantially more compact than conjunctive or disjunctive normal form, and very efficient algorithms have been developed for manipulating them. Because the symbolic representation captures some of the regularity in the state space determined by circuits and protocols, it is possible to verify systems with an extremely large number of states—many orders of magnitude larger than could be handled by the explicit-state algorithms. By using the original CTL model checking algorithm [61] of Clarke and Emerson with the new representation for state transition graphs, it became possible to verify some examples that had more than 10^{20} states [46, 191]. Since then, various refinements of the OBDD-based techniques by other researchers have pushed the state count up to more than 10^{120} [43, 44].

The implicit representation is quite natural for modeling sequential circuits and protocols. Each state is encoded by an assignment of boolean values to the set of state variables associated with the circuit or protocol. The transition relation can therefore be expressed as a boolean formula in terms of two sets of variables, one set encoding the old state and the other encoding the new. This formula is then represented by a binary decision diagram. The model checking algorithm is based on computing fixpoints of *predicate transformers* that are obtained from the transition relation. The fixpoints are sets of states that represent various temporal properties of the concurrent system. In the new implementations, both the predicate transformers and the fixpoints are represented with OBDDs. Thus, it is possible to avoid explicitly constructing the state graph of the concurrent system.

The model checking system that McMillan developed as part of his doctoral dissertation thesis is called SMV [191]. It is based on a language for describing hierarchical finite-state concurrent systems. Programs in the language can be annotated by specifications expressed in temporal logic. The model checker extracts a transition system represented as an OBDD from a program in the SMV language and uses an OBDD-based search algorithm to determine whether the system satisfies its specification. If the transition system does not satisfy some specification, the verifier will produce an execution trace that shows why the specification is false. The SMV system has been widely distributed, and a large number of examples have now been verified with it. These examples provide convincing evidence that SMV can be used to debug real industrial designs.

An impressive example that illustrates the power of symbolic model checking is the verification of the cache coherence protocol described in the IEEE Futurebus+ standard (IEEE Standard 896.1-1991). Although development of the Futurebus+ cache coherence protocol began in 1988, all previous attempts to validate the protocol were based entirely on informal techniques. In the summer of 1992 researchers at Carnegie Mellon [66, 179]

constructed a precise model of the protocol in SMV language and then used SMV to show that the resulting transition system satisfied a formal specification of cache coherence. They were able to find a number of previously undetected errors and potential errors in the design of the protocol. This appears to be the first time that an automatic verification tool has been used to find errors in an IEEE standard.

One of the best indications of the power of the symbolic verification methods comes from studying how the CPU time required for verification grows asymptotically with larger and larger instances of the circuit or protocol. In many of the examples that have been considered by a variety of groups, this growth rate is a small polynomial in the number of components of the circuit [18, 43, 44].

A number of other researchers have independently discovered that OBDDs can be used to represent large state-transition systems. Coudert, Berthet, and Madre [81] have developed an algorithm for showing equivalence between two deterministic finite-state automata by performing a breadth first search of the state space of the product automata. They use OBDDs to represent the transition functions of the two automata in their algorithm. Similar algorithms have been developed by Pixley [213, 214, 215]. In addition, several groups including Bose and Fisher [26], Pixley [213], and Coudert, Madre, and Berthet [82] have experimented with model checking algorithms that use OBDDs.

In related work Bryant, Seger and Beatty [18, 37] have developed an algorithm based on symbolic simulation for model checking in a restricted linear time logic. Specifications consist of precondition–postcondition pairs expressed in the logic. The precondition is used to restrict inputs and initial states of the circuit; the postcondition gives the property that the user wishes to check. Formulas in the logic have the form

$$p_0 \wedge \mathbf{X}p_1 \wedge \mathbf{X}^2 p_2 \wedge \cdots \wedge \mathbf{X}^{n-1} p_{n-1} \wedge \mathbf{X}^n p_n.$$

The syntax of the formulas is highly restricted compared to most other temporal logics used for specifying programs and circuits. In particular, the only logical operator that is allowed is conjunction, and the only temporal operator is *next time* (**X**). By limiting the class of formulas that can be handled, it is possible to check certain properties very efficiently.

1.6 Partial Order Reduction

Verifying software causes some problems for model checking. Software tends to be less structured than hardware. In addition, concurrent software is usually *asynchronous*, that is, most of the activities taken by different processes are performed independently, without a global synchronizing clock. For these reasons, the state explosion phenomenon is a particularly serious problem for software. Consequently, model checking has been used less

frequently for software verification than for hardware verification. Recently, considerable progress has been made on the state explosion problem for software. The most successful techniques for dealing with this problem are based on the *partial order reduction* [126, 209, 244]. These techniques exploit the independence of concurrently executed events. Two events are *independent* of each other when executing them in either order results in the same global state.

A common model for representing concurrent software is the *interleaving model*, in which all of the events in a single execution are arranged in a linear order called an *interleaving sequence*. Concurrently executed events appear arbitrarily ordered with respect to one another. Most logics for specifying properties of concurrent systems can distinguish between interleaving sequences in which two independent events are executed in different orders. Because of this, all possible interleavings of such events are normally considered. This can result in an extremely large state space.

The partial order reduction techniques make it possible to decrease the number of interleaving sequences that must be considered. As a result, the number of states that are needed for model checking is reduced. When a specification cannot distinguish between two interleaving sequences that differ only by the order in which concurrently executed events are taken, it is sufficient to analyze only one of them. These methods are related to the *partial order model of program execution*. According to this model, concurrently executed events are not ordered. Each partially ordered execution can correspond to multiple interleaving sequences. If it is impossible to distinguish between such sequences, it is sufficient to select one interleaving sequence for each partial ordering of events.

The idea of reducing the state space by selecting only a subset of the ways one can interleave independently executed transitions has been studied by many researchers. One of the first researchers to propose such a reduction technique was Overman [205]. However, he only considered a restricted model of concurrency that did not include looping and nondeterministic choice. The proof system of Katz and Peled [153] suggests using an equivalence relation between interleaving sequences that correspond to the same partially ordered execution. Their system includes proof rules for reasoning about a selection of interleaved sequences rather than all of them. Model checking algorithms that incorporate the partial order reduction are described in several different papers. The *stubborn sets* of Valmari [244], the *persistent sets* of Godefroid [125] and the *ample sets* of Peled [209] differ on the actual details, but contain many similar ideas. In this book we will describe the ample set method. Other methods that exploit similar observations about the relation between the partial and total order models of execution are McMillan's *unfolding technique* [190] and Godefroid's *sleep sets* [125].

1.7 Other Approaches to the State Explosion Problem

Although symbolic representations and the partial order reduction have greatly increased the size of the systems that can be verified, many realistic systems are still too large to be handled. Thus, it is important to find techniques that can be used in conjunction with the symbolic methods to extend the size of the systems that can be verified. Four such techniques are compositional reasoning, abstraction, symmetry, and induction.

The first technique exploits the *modular structure* of complex circuits and protocols [72, 128, 129, 150, 151, 168, 218, 230]. Many finite state systems are composed of multiple processes running in parallel. The specifications for such systems can often be decomposed into properties that describe the behavior of small parts of the system. An obvious strategy is to check each of the local properties, using only the part of the system that it describes. If it is possible to show that the system satisfies each local property, and if the conjunction of the local properties implies the overall specification, then the complete system must satisfy this specification as well.

When this naive form of compositional reasoning is not feasible because of mutual dependencies between the components, a more complex strategy is necessary. In such cases, when verifying a property of one component we must make assumptions about the behavior of the other components. The assumptions must later be discharged when the correctness of the other components is established. This strategy is called *assume-guarantee reasoning* [129, 150, 151, 198, 218].

The second technique involves using *abstraction*. This technique appears to be essential for reasoning about reactive systems that involve data paths. Traditionally, finite state verification methods have been used mainly for control-oriented systems. The symbolic methods make it possible to handle some systems that involve nontrivial data manipulation, but the complexity of verification is often high. The use of abstraction is based on the observation that the specifications of systems that include data paths usually involve fairly simple relationships among the data values in the system. For example, in verifying the addition operation of a microprocessor, we might require that the value in one register is eventually equal to the sum of the values in two other registers. In such situations *abstraction* can be used to reduce the complexity of model checking [20, 69, 90, 91, 160, 248]. The abstraction is usually specified by giving a mapping between the actual data values in the system and a small set of abstract data values. By extending the mapping to states and transitions, it is possible to produce an abstract version of the system under consideration. The abstract system is often much smaller than the actual system, and as a result, it is usually much simpler to verify properties at the abstract level.

Symmetry can also be used to reduce the state explosion problem [64, 111, 143, 148]. Finite state concurrent systems frequently contain replicated components. For example,

a large number of protocols involve a network of identical processes communicating in some fashion. Hardware devices contain parts such as memories and register files that have many replicated elements. These facts can be used to obtain reduced models for the system. Having symmetry in a system implies the existence of a nontrivial permutation group that preserves the state transition graph. Such a group can be used to define an equivalence relation on the state space of the system and to reduce the state space. The reduced model can be used to simplify the verification of properties of the original model expressed by a temporal logic formula.

Induction involves reasoning automatically about entire families of finite-state systems [33, 67, 155, 165, 187, 229, 250]. Such families arise frequently in the design of reactive systems in both hardware and software. Typically, circuit and protocol designs are parameterized, that is, they define an infinite family of systems. For example, a circuit designed to add two integers has the width of the integers n as a parameter; a bus protocol may be designed to accommodate an arbitrary number of processors, and a mutual exclusion protocol can be given for a parameterized number of processes. We would like to be able to check that every system in a given family satisfies some temporal logic property. In general the problem is undecidable [12, 237], but in many interesting cases, it is possible to provide a form of *invariant process* that represents the behavior of an arbitrary member of the family. Using this invariant, we can then check the property for all of the members of the family at once. An inductive argument is used to verify that the invariant is an appropriate representative.

2 Modeling Systems

The first step in verifying the correctness of a system is specifying the properties that the system should have. For example, we may want to show that some concurrent program never deadlocks. Once we know which properties are important, the second step is to construct a *formal model* for the system. In order to be suitable for verification, the model should capture those properties that must be considered to establish correctness. On the other hand, it should abstract away those details that do not effect the correctness of the checked properties but make verification more complicated. For example, when modeling digital circuits, it is useful to reason in terms of gates and boolean values, rather than actual voltage levels. Likewise, when reasoning about a communication protocol we may want to focus on the exchange of messages and ignore the actual contents of the messages.

In this book, we will be primarily concerned with *reactive systems* [186] and their behavior over time. Such systems may need to interact with their environment frequently and often do not terminate. Therefore, they cannot adequately be modeled by their input-output behavior. The first feature of a reactive system that we want to capture is its *state*. A state is a snapshot or instantaneous description of the system that captures the values of the variables at a particular instant of time. We also need to know how the state of the system changes as the result of some action of the system. We can describe the change by giving the state before the action occurs and the state after the action occurs. Such a pair of states determines a *transition* of the system. The computations of a reactive system can be defined in terms of its transitions. A *computation* is an infinite sequence of states where each state is obtained from the previous state by some transition.

We use a type of state transition graph called a *Kripke structure* to capture this intuition about the behavior of reactive systems. A Kripke structure consists of a set of states, a set of transitions between states, and a function that labels each state with a set of properties that are true in this state. Paths in a Kripke structure model computations of the system. Although these models are very simple, they are sufficiently expressive to capture those aspects of temporal behavior that are most important for reasoning about reactive systems.

Concurrent systems are usually given by the text of a program or by the diagram for a circuit. There are many different types of concurrent systems (synchronous and asynchronous circuits, programs with shared variables, programs that communicate by message passing, and so on). Because of this diversity we need a unifying formalism that can represent a concurrent system of any type. We will use formulas of first order logic for this purpose. Given a formula that represents a concurrent system, it is straightforward to extract the Kripke structure that models the system.

In the following sections we formally define Kripke structures. We show how to extract such structures from first order formulas that represent concurrent systems. Finally, we demonstrate how different programming constructs can be represented in terms of first order formulas.

2.1 Modeling Concurrent Systems

Let AP be a set of atomic propositions. A *Kripke structure* M over AP is a four tuple $M = (S, S_0, R, L)$ where

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

Sometimes we will not be concerned with the set of initial states S_0. In such cases, we will omit this set of states from the definition. A *path* in the structure M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

2.1.1 First Order Representations

We only assume a basic knowledge of first order logic. The reader should be familiar with the logical connectives (*and* $\wedge$, *or* $\vee$, *not* $\neg$, *implies* $\rightarrow$, and so on) and should know how universal ($\forall$) and existential ($\exists$) quantification work.

We use interpreted first order formulas to describe concurrent systems. Thus, the predicate and function symbols that occur in such formulas will have a predefined meaning. Usually, this meaning will be clear from the context. Let $V = \{v_1, \ldots, v_n\}$ be the set of system variables. We assume that the variables in V range over a finite set D (sometimes called the *domain* or *universe* of the interpretation). A *valuation* for V is a function that associates a value in D with each variable v in V.

A *state* of a concurrent system can be described by giving values for all of the elements in V. In other words, a state is just a valuation $s : V \to D$ for the set of variables in V. Given a valuation, we can write a formula that is true for exactly that valuation. For example, given $V = \{v_1, v_2, v_3\}$ and the valuation $\langle v_1 \leftarrow 2, v_2 \leftarrow 3, v_3 \leftarrow 5 \rangle$, we derive the formula $(v_1 = 2) \wedge (v_2 = 3) \wedge (v_3 = 5)$. In general, a formula may be true for many valuations. If we adopt the convention that a formula represents the set of *all* valuations that make it true, then we can describe certain sets of states by first order formulas. In particular, the *initial states* of the system can be described by a first order formula $\mathcal{S}_0$ over the variables in V.

In addition to representing sets of states, we must be able to represent sets of transitions between states. To do this, we extend the idea used above. This time, we use a formula to represent a set of ordered pairs of states. We cannot do this using just a single copy of the

system variables V, so we create a second set of variables V'. We think of the variables in V as *present state* variables and the variables in V' as *next state* variables. Each variable v in V has a corresponding next state variable in V', which we denote by v'. A valuation for the variables in V and V' can be viewed as designating an ordered pair of states or a transition, and we can represent sets of these valuations using formulas as above. We refer to a set of pairs of states as a *transition relation*. If R is a transition relation, then we write $\mathcal{R}(V, V')$ to denote a formula that represents it.

In order to write specifications that describe properties of concurrent systems we need to define a set of atomic propositions AP. Atomic propositions will typically have the form $v = d$ where $v \in V$ and $d \in D$. A proposition $v = d$ will be true in a state s if $s(v) = d$. When v is a variable over the boolean domain $\{True, False\}$, it is not necessary to include both $v = True$ and $v = False$ in AP. We will write v to indicate that $s(v) = True$ and $\neg v$ to indicate that $s(v) = False$.

We now show how to derive a Kripke structure $M = (S, S_0, R, L)$ from the first order formulas $\mathcal{S}_0$ and $\mathcal{R}$ that represent the concurrent system.

- The set of states S is the set of all valuations for V.
- The set of initial states S_0 is the set of all valuations s_0 for V that satisfy the formula $\mathcal{S}_0$.
- Let s and s' be two states, then $R(s, s')$ holds if $\mathcal{R}$ evaluates to *True* when each $v \in V$ is assigned the value $s(v)$ and each $v' \in V'$ is assigned the value $s'(v)$.
- The labeling function $L : S \to 2^{AP}$ is defined so that $L(s)$ is the subset of all atomic propositions true in s. If v is a variable over the boolean domain, then $v \in L(s)$ indicates that $s(v) = True$, and $v \notin L(s)$ indicates that $s(v) = False$.

Because we require that the transition relation of a Kripke structure is always total, we must extend the relation R if some state s has no successor. In this case, we modify R so that $R(s, s)$ holds.

To illustrate the notions defined in this section we consider a simple system with variables x and y that range over $D = \{0, 1\}$. Thus, a valuation for the variables x and y is just a pair $(d_1, d_2) \in D \times D$ where d_1 is the value for x and d_2 is the value for y. The system consists of one transition

$$x := (x + y) \bmod 2,$$

which starts from the state in which $x = 1$ and $y = 1$. This system will be described by two first order formulas. The set of initial states of the system is represented by

$$\mathcal{S}_0(x, y) \equiv x = 1 \wedge y = 1,$$

and the set of transitions is represented by

$\mathcal{R}(x, y, x', y') \equiv x' = (x + y) \bmod 2 \wedge y' = y.$

The Kripke structure $M = (S, S_0, R, L)$ extracted from these formulas is:

- $S = D \times D$.
- $S_0 = \{(1, 1)\}$.
- $R = \{((1, 1), (0, 1)), ((0, 1), (1, 1)), ((1, 0), (1, 0)), ((0, 0), (0, 0))\}$.
- $L((1, 1)) = \{x = 1, y = 1\}$, $L((0, 1)) = \{x = 0, y = 1\}$, $L((1, 0)) = \{x = 1, y = 0\}$, and $L((0, 0)) = \{x = 0, y = 0\}$.

The only path in the Kripke structure that starts in an initial state is (1,1) (0,1) (1,1) (0,1) . . . . This path is the only computation of the system.

2.1.2 Granularity of Transitions

A critical issue in modeling concurrent systems is determining the granularity of the transitions. It is important to obtain transitions that are *atomic* in the sense that no observable state of the system can result from executing part of a transition. A common mistake is to define transitions that are too coarse. In this case, the Kripke structure may not include some states that are observable. As a result, verification techniques such as model checking may fail to find important errors. A problem can also arise when the granularity is too fine. In this case transitions can interact to create new states that are not reachable in the actual system. As a result, model checking may find spurious errors that will never occur in practice.

For an example, consider a system with two variables x and y and two transitions α and β that can be executed concurrently.

α: $x := x + y$ and

β: $y := y + x$

with the initial state $x = 1 \wedge y = 2$. Also consider a finer grained implementation of the same transitions. This implementation uses the assembly language instructions for loading, adding, and storing between a memory address and a register:

α_0: load R_1, x	β_0: load R_2, y
α_1: add R_1, y	β_1: add R_2, x
α_2: store R_1, x	β_2: store R_2, y

Executing α and then β results in the state $x = 3 \wedge y = 5$. When β is executed before α, we obtain $x = 4 \wedge y = 3$. If, on the other hand, the finer grained implementation is executed in the order $\alpha_0\beta_0\alpha_1\beta_1\alpha_2\beta_2$, the result is $x = 3 \wedge y = 3$.

Suppose that $x = 3 \wedge y = 3$ violates some desired property of the system. Further suppose that the system is implemented using the transitions α and β. Then, it is impossible to have $x = 3$ and $y = 3$ at the same time. However, if we model the system with the finer grained transitions $\alpha_0, \alpha_1, \alpha_2, \beta_0, \beta_1$, and β_2, we may erroneously conclude that the system is incorrect. Next, suppose that the system is implemented using $\alpha_0, \alpha_1, \alpha_2, \beta_0, \beta_1$, and β_2. In this case it is possible to reach a state in which both $x = 3$ and $y = 3$. If we now model the system with α and β, we will erroneously conclude that the system is correct.

Extracting a first order representation from the text of a program or a diagram of a circuit can be viewed as a compilation task. This task must take into account granularity considerations like the one described above. In the next section, we will discuss in greater detail how the compilation is performed.

2.2 Concurrent Systems

A concurrent system consists of a set of components that execute together. Normally the components have some means of communicating with each other. The mode of execution and the mode of communication may differ from one system to another. We will consider two modes of execution: *Asynchronous* or *interleaved execution*, in which only one component makes a step at a time, and *synchronous execution*, in which all of the components make a step at the same time. We will also distinguish three modes of communication. Components can either communicate by changing the values of *shared variables* or by *exchanging messages* using queues or some type of handshaking protocol. Because modeling is not the main concern of this book, we will only discuss communication by shared variables in this chapter.

In the following sections we describe some important types of concurrent systems and show how they can be represented in terms of first order formulas. From these formulas we can derive Kripke structures for the systems, as shown in Section 2.1.1.

2.2.1 Digital Circuits

In this section, we show how to describe circuits by formulas. For simplicity, we assume that each *state holding element* of a circuit can have the value 0 or 1. Let V be the set of state holding elements of a circuit. For a synchronous circuit, the set V typically consists of the outputs of all the registers in the circuit together with the primary inputs. For asynchronous circuits, all wires in the circuit are usually considered to be state holding elements. If we create a boolean variable for each element in V, then a state can be described by a valuation assigning either 0 or 1 to each variable. Given a valuation, we can write a boolean expression that is true for exactly that valuation. For example, given

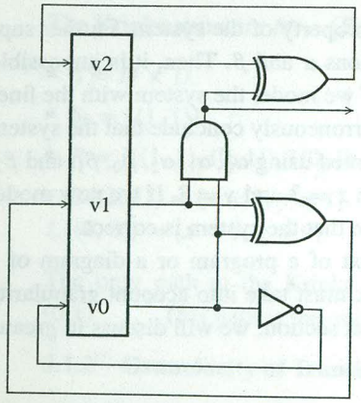


Figure 2.1
Synchronous modulo 8 counter.

$V = \{v_1, v_2\}$ and the valuation $\langle v_1 \leftarrow 1, v_2 \leftarrow 0\rangle$, we derive the boolean formula $v_1 \wedge \neg v_2$. As before, we adopt the convention that a formula represents the set of *all* valuations that make it true. Thus, for describing circuits the full expressive power of first order logic is not needed; boolean formulas are sufficient. The boolean formulas $\mathcal{S}_0(V)$ and $\mathcal{R}(V, V')$ will represent the set of initial states and the transition relation of the circuit, respectively.

Synchronous Circuits

The operation of a synchronous circuit consists of a sequence of steps. In each step, the inputs to the circuit change and the circuit is allowed to stabilize. Then a clock pulse occurs, and the state-holding elements change.

The method for deriving the transition relation of a synchronous circuit can be illustrated using a small example. The circuit in Figure 2.1 is a modulo 8 counter. Let $V = \{v_0, v_1, v_2\}$ be the set of state variables for this circuit, and let $V' = \{v_0', v_1', v_2'\}$ be another copy of the state variables. The transitions of the modulo 8 counter are given by

$$v_0' = \neg v_0$$

$$v_1' = v_0 \oplus v_1$$

$$v_2' = (v_0 \wedge v_1) \oplus v_2$$

where $\oplus$ is the *exclusive or* operator. The above equations can be used to define the relations

$$\mathcal{R}_0(V, V') \equiv (v_0' \Leftrightarrow \neg v_0)$$

$$\mathcal{R}_1(V, V') \equiv (v_1' \Leftrightarrow v_0 \oplus v_1)$$

$$\mathcal{R}_2(V, V') \equiv (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2)$$

which describe the constraints each v_i' must satisfy in a legal transition. Because all the changes occur at the same time, the constraints are combined by taking their conjunction to construct a formula for the transition relation:

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \wedge \mathcal{R}_1(V, V') \wedge \mathcal{R}_2(V, V').$$

In the general case of a synchronous circuit with n state holding elements, we let $V = \{v_0, \cdots, v_{n-1}\}$ and $V' = \{v_0', \ldots, v_{n-1}'\}$. Analogous to the modulo 8 counter, for each state variable v_i' there is a boolean function f_i such that

$$v_i' = f_i(V).$$

These equations are used to define the relations

$$\mathcal{R}_i(V, V') \equiv (v_i' \Leftrightarrow f_i(V)).$$

Continuing the analogy with the modulo 8 counter, the conjunction of these formulas forms the transition relation

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \wedge \cdots \wedge \mathcal{R}_{n-1}(V, V').$$

Thus, the transition relation for a synchronous circuit can be expressed as the conjunction of the transition relations of the individual processes.

Asynchronous Circuits

The transition relation for an asynchronous circuit is most naturally expressed as a disjunction. To simplify the description of how the transition relations are obtained, we assume that all the components of the circuits have exactly one output and have no internal state variables. In this case, it is possible to describe each component by a function $f_i(V)$; given values for the present state variables v, the component drives its output to the value specified by $f_i(V)$. Extending the method to handle components with multiple outputs is straightforward.

Because the value of a component can change so rapidly, it is unlikely that two components will change at the same time. For this reason, it is customary to use an *interleaving semantics* in which exactly one component changes at a time. This results in a disjunction of the form:

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \vee \cdots \vee \mathcal{R}_{n-1}(V, V'),$$

where

$$\mathcal{R}_i(V, V') \equiv \left(v_i' \Leftrightarrow f_i(V)\right) \wedge \bigwedge_{j \neq i} (v_j' \Leftrightarrow v_j).$$

Note that some component may change repeatedly, without another component ever making a step. In practice, this is extremely unlikely. It is possible to augment the model with an additional *fairness* constraint that will disallow such behaviors. This topic will be discussed further in the next chapter.

To illustrate the difference between the synchronous and the asynchronous models, consider the following example. Let $V = \{v_0, v_1\}$, $v_0' = v_0 \oplus v_1$ and $v_1' = v_0 \oplus v_1$. Let s be a state with $v_0 = 1 \wedge v_1 = 1$. According to the synchronous model, the only successor of s is the state with $v_0 = 0 \wedge v_1 = 0$, since both assignments are executed simultaneously. According to the asynchronous model, the state s has two successors:

1. $v_0 = 0 \wedge v_1 = 1$ (the assignment to v_0 is taken first).
2. $v_0 = 1 \wedge v_1 = 0$ (the assignment to v_1 is taken first).

2.2.2 Programs

All of the programs we consider are asynchronous. We start by discussing sequential programs because concurrent programs are composed of sequential components. The approach that we use is similar to the approach used in the book by Manna and Pnueli [186]. For a more detailed treatment of these issues, we refer the reader to that book. A program consists of statements that are sequentially composed with each other. We describe a translation procedure $\mathcal{C}$ that takes the text of a sequential program P and transforms it into a first order formula $\mathcal{R}$ that represents the set of transitions of the program. Without loss of generality, we assume that each statement has a unique *entry point* and a unique *exit point*. The transition procedure is simplified significantly if each entry and exit point of a statement in the program is uniquely labeled. Thus, we define a labeling transformation that given an unlabeled program P results in a labeled program $P^{\mathcal{L}}$.

The labeling transformation defined below attaches a single label with the entry point of each statement in P, except for P itself. No two attached labels are identical. In sequential programs, the exit point of a statement is identical to the entry point of the following statement. Thus, it is sufficient to label entry points. If we also provide labels for the entry and the exit points of P, then we get a unique labeling of the entry and exit points of all statements of the program.

Since we do not restrict ourselves to a specific programming language, we define the labeling transformation for a number of statement types. It is easy to extend the definition to other statement types. Given a statement P, the *labeled statement* $P^{\mathcal{L}}$ is defined as follows:

- If P is not a composite statement (e.g., P is $x := e$, **skip**, **wait**, **lock**, **unlock**, etc.), then $P^{\mathcal{L}} = P$.
- If $P = P_1; P_2$ then $P^{\mathcal{L}} = P_1^{\mathcal{L}}; l'' : P_2^{\mathcal{L}}$.
- If $P =$ **if** b **then** P_1 **else** P_2 **end if**, then $P^{\mathcal{L}} =$ **if** b **then** $l_1 : P_1^{\mathcal{L}}$ **else** $l_2 : P_2^{\mathcal{L}}$ **end if**.
- If $P =$ **while** b **do** P_1 **end while**, then $P^{\mathcal{L}} =$ **while** b **do** $l_1 : P_1^{\mathcal{L}}$ **end while**.

In the remainder of this section, we assume that P is a labeled statement and that the entry and exit points of P are labeled by m and m' respectively. Let pc be a special variable called the *program counter* that ranges over the set of program labels and an additional value $\perp$ called the *undefined value*. The undefined value is needed when concurrent programs are considered. In this case, $pc = \perp$ indicates that the program is not active.

Let V denote the set of program variables. Let V' be the set of primed variables v' for each $v \in V$, and let pc' be the primed variable for pc. Recall that the unprimed copy refers to the value of the variables before a transition, whereas the primed copy refers to the value after the transition. Because each transition typically changes only a small number of the program variables, we will use $same(Y)$ as an abbreviation for the formula

$$\bigwedge_{y \in Y} (y' = y).$$

We first give the formula that describes the set of initial states of the program P. Given some condition $pre(V)$ on the initial values of the variables of P,

$$\mathcal{S}_0(V, pc) \equiv pre(V) \wedge pc = m.$$

The translation procedure $\mathcal{C}$ depends on three parameters: the entry label l, the labeled statement P, and the exit label l'. The procedure is defined recursively with one rule for each statement type in the language. $\mathcal{C}(l, P, l')$ describes the set of transitions in P as a disjunction of all the transitions in the set. The disjunct for an individual transition determines the value of the boolean condition and the value of the program counter for which the transition may be executed. It is true whenever the transition is enabled and false otherwise.

- Assignment:

$$\mathcal{C}(l, v \leftarrow e, l') \equiv pc = l \wedge pc' = l' \wedge v' = e \wedge same(V \setminus \{v\})$$

- Skip:

$$\mathcal{C}(l, skip, l') \equiv pc = l \wedge pc' = l' \wedge same(V)$$

- Sequential composition:

$\mathcal{C}(l,\ P_1;\ l'' : P_2,\ l') \equiv \mathcal{C}(l,\ P_1,\ l'') \ \vee\ \mathcal{C}(l'',\ P_2,\ l')$

The formula for the transitions of $P_1;\ l'' : P_2$ is a disjunction of the formulas for the transitions of P_1 and of P_2. Because of the intermediate label l'', statement P_2 will be executed after statement P_1.

- Conditional:

$\mathcal{C}(l,$ **if** b **then** $l_1 : P_1$ **else** $l_2 : P_2$ **end if**, $l')$ is the disjunction of the following four formulas:

- $pc = l \ \wedge\ pc' = l_1 \ \wedge\ b \ \wedge\ same(V)$
- $pc = l \ \wedge\ pc' = l_2 \ \wedge\ \neg b \ \wedge\ same(V)$
- $\mathcal{C}(l_1,\ P_1,\ l')$
- $\mathcal{C}(l_2,\ P_2,\ l')$

The first disjunct corresponds to the case where condition b is true. In this case, statement P_1 will be executed next. The second disjunct corresponds to the case where condition b is false. In this case, statement P_2 will be executed next. Both disjuncts describe transitions that involve only a change of the program counter. The third and fourth disjuncts are formulas for the transitions of P_1 and P_2, respectively. Note that l' is the exit point for both P_1 and P_2. The translation for the **if** statement can easily be extended to handle nondeterministic choice between several alternatives.

- While:

$\mathcal{C}(l,$ **while** b **do** $l_1 : P_1$ **end while**, $l')$ is the disjunction of the following three formulas:

- $pc = l \ \wedge\ pc' = l_1 \ \wedge\ b \ \wedge\ same(V)$
- $pc = l \ \wedge\ pc' = l' \ \wedge\ \neg b \ \wedge\ same(V)$
- $\mathcal{C}(l_1,\ P_1,\ l)$

The first disjunct corresponds to the case where condition b is true. In this case, statement P_1 will be executed next. The second disjunct corresponds to the case where condition b is false, in which case, the execution of the while statement terminates. The third disjunct is a formula for the set of transitions of P_1. Note that the exit point of P_1 is identical to the entry point of the while statement. Thus, if P_1 terminates the execution of the while statement will restart.

2.2.3 Concurrent Programs

A *concurrent program* consists of a set of processes that can be executed in parallel. A *process* is a sequential statement as described in the previous section. Concurrent programs in which processes do not interact by means of message passing or shared variables are

usually easy to analyze and will not be considered further. In this section, we will consider asynchronous programs in which exactly one process can make a transition at any time. We begin by introducing some terminology that will be used throughout the section. V_i is the set of variables that can be changed by process P_i. We do not require that these sets be disjoint. As before, V is the set of all program variables. The program counter of a process P_i is pc_i. PC is the set of all program counters.

A concurrent program P has the form

cobegin $P_1 \| P_2 \| \ldots \| P_n$ **coend**

where $P_1, \ldots, P_n$ are processes. The labeling transformation for sequential programs is extended so that a concurrent program can occur as a statement in a sequential program. The transformation attaches a label to the entry point and to the exit point of each process. Unlike exit points in sequential programs, no exit point of a concurrent process is identical to an entry point. As a result, the exit points of processes must be explicitly labeled. As before, we assume that no two labels are identical and that the entry and exit points of P are labeled m and m', respectively.

- If $P =$ **cobegin** $P_1 \| P_2 \| \ldots \| P_n$ **coend**, then
$P^{\mathcal{L}} =$ **cobegin** $l_1 : P_1^{\mathcal{L}}\ l_1' \| l_2 : P_2^{\mathcal{L}}\ l_2' \| \ldots \| l_n : P_n^{\mathcal{L}}\ l_n'$ **coend**.

The formula that describes the initial states of a concurrent program P is

$$\mathcal{S}_0(V, PC) \equiv pre(V) \wedge pc = m \wedge \bigwedge_{i=1}^{n} (pc_i = \bot),$$

where $pc_i = \bot$ indicates that process P_i has not been activated yet and therefore cannot be executed from the current state.

The translation procedure $\mathcal{C}$ is extended to concurrent programs as follows: $\mathcal{C}(l,$ **cobegin** $l_1 : P_1\ l_1' \| \ldots \| l_n : P_n\ l_n'$ **coend**, $l')$ is the disjunction of three formulas:

- $pc = l \ \wedge\ pc_1' = l_1 \ \wedge\ \ldots \ \wedge\ pc_n' = l_n \ \wedge\ pc' = \bot$
- $pc = \bot \ \wedge\ pc_1 = l_1' \ \wedge\ \ldots \ \wedge\ pc_n = l_n' \ \wedge\ pc' = l' \ \wedge\ \bigwedge_{i=1}^{n} (pc_i' = \bot)$
- $\bigvee_{i=1}^{n} \big(\mathcal{C}(l_i, P_i, l_i') \ \wedge\ same(V \setminus V_i) \ \wedge\ same(PC \setminus \{ pc_i \})\big)$

The first disjunct describes the initialization of the concurrent processes. A transition is made from the entry point of the **cobegin** statement to the entry points of the individual processes. The second disjunct describes the termination of the concurrent program. A transition is made from the exit points of the processes to the exit of the **cobegin** statement. This transition will only be executed if all the processes terminate. The third disjunct

describes the interleaved execution of the concurrent processes. The formula for the transition relation of process P_i is conjuncted with $same\ (V \setminus V_i)\ \wedge same(PC \setminus \{\ pc_i\ \}))$. This guarantees that a transition in process P_i can only change variables in V_i. It also ensures that only one process can make a transition at any time.

Shared Variables

Recall that V_i is the set of variables that may be changed by process P_i. Concurrent programs for which the sets V_i overlap are called *shared variable* programs. We show how to extend the translation procedure $\mathcal{C}$ to some commonly used *process synchronization* statements. Such statements are frequently needed to provide processes with exclusive access to shared variables. These statements are atomic and treated by the labeling transformation accordingly. Assume that the statement belongs to the text of process P_i.

- Wait: Because our primary interest is in finite state programs, we only describe how to implement this statement using *busy waiting*. In particular, we do not consider implementations that require complex data structures like process queues. The statement **wait**(b) repeatedly tests the value of the boolean variable b until it determines that b is true. When b becomes true, a transition is made to the next program point

  $\mathcal{C}(l,\ \textbf{wait}(b),\ l')$ is a disjunction of the following two formulas:

  - $(pc_i = l\ \wedge\ pc_i' = l\ \wedge\ \neg b\ \wedge\ same(V_i))$
  - $(pc_i = l\ \wedge\ pc_i' = l'\ \wedge\ b\ \wedge\ same(V_i))$

- Lock: The statement **lock**(v) is similar to the statement **wait**$(v = 0)$, except that when $v = 0$ is true the transition changes the value of v to 1. This statement is often used to guarantee *mutual exclusion* by preventing more than one process from entering its critical region.

  $\mathcal{C}(l,\ \textbf{lock}(v),\ l')$ is a disjunction of the following two formulas:

  - $(pc_i = l\ \wedge\ pc_i' = l\ \wedge\ v = 1\ \wedge\ same(V_i))$
  - $(pc_i = l\ \wedge\ pc_i' = l'\ \wedge\ v = 0\ \wedge\ v' = 1\ \wedge same(V_i \setminus \{v\}))$

- Unlock: The statement **unlock**(v) assigns the value 0 to the variable v. Typically, this statement enables some other process to enter its critical region.

$\mathcal{C}(l,\ \textbf{unlock}(v),\ l') \equiv\ pc_i = l\ \wedge\ pc_i' = l'\ \wedge\ v' = 0\ \wedge\ same(V_i \setminus \{v\})$

2.3 Example of Program Translation

Consider a simple *mutual exclusion* program

$P = m$: **cobegin** $P_0 \parallel P_1$ **coend** m'

with two processes P_0 and P_1, where

$P_0 ::\ l_0$: **while** *True* **do**
NC_0 : **wait**$(turn = 0)$;
CR_0 : $turn := 1$;
end while;
l_0'

$P_1 ::\ l_1$: **while** *True* **do**
NC_1 : **wait**$(turn = 1)$;
CR_1 : $turn := 0$;
end while;
l_1'

The program counter pc of the program P takes only three values: m, the label of the entry point of P; m', the label of the exit point of P; and $\perp$ the value of pc when P_1 and P_2 are active. Each process P_i has a program counter pc_i that ranges over the labels l_i, l_i', NC_i, CR_i, and $\perp$. The two processes share a single variable $turn$. Thus, $V = V_0 = V_1 = \{turn\}$ and $PC = \{pc, pc_0, pc_1\}$. When the value of the program counter of a process P_i is CR_i, the process is in its *critical region*. Both processes are not allowed to be in their critical regions at the same time. When the value of the program counter is NC_i, the process is in its *noncritical region*. In this case it waits until $turn = i$ in order to gain exclusive entry into the critical region.

The initial states of P are described by the formula

$S_0(V, PC) \equiv pc = m\ \wedge\ pc_0 = \perp\ \wedge\ pc_1 = \perp.$

Note that no restriction is imposed on the value of $turn$. Thus, it may initially be either 0 or 1. Applying the translation procedure $\mathcal{C}$ we obtain the formula for the transition relation of P, $\mathcal{R}(V, PC, V', PC')$, which is the disjunction of the following four formulas:

- $pc = m\ \wedge\ pc_0' = l_0\ \wedge\ pc_1' = l_1\ \wedge\ pc' = \perp$
- $pc_0 = l_0'\ \wedge\ pc_1 = l_1'\ \wedge\ pc' = m'\ \wedge\ pc_0' = \perp\ \wedge\ pc_1' = \perp$
- $\mathcal{C}(l_0, P_0, l_0')\ \wedge\ same(V \setminus V_0)\ \wedge\ same(PC \setminus \{\ pc_0\ \})$, which is equivalent to $\mathcal{C}(l_0, P_0, l_0')\ \wedge\ same(pc, pc_1)$
- $\mathcal{C}(l_1, P_1, l_1')\ \wedge\ same(V \setminus V_1)\ \wedge\ same(PC \setminus \{\ pc_1\ \})$, which is equivalent to $\mathcal{C}(l_1, P_1, l_1')\ \wedge\ same(pc, pc_0)$

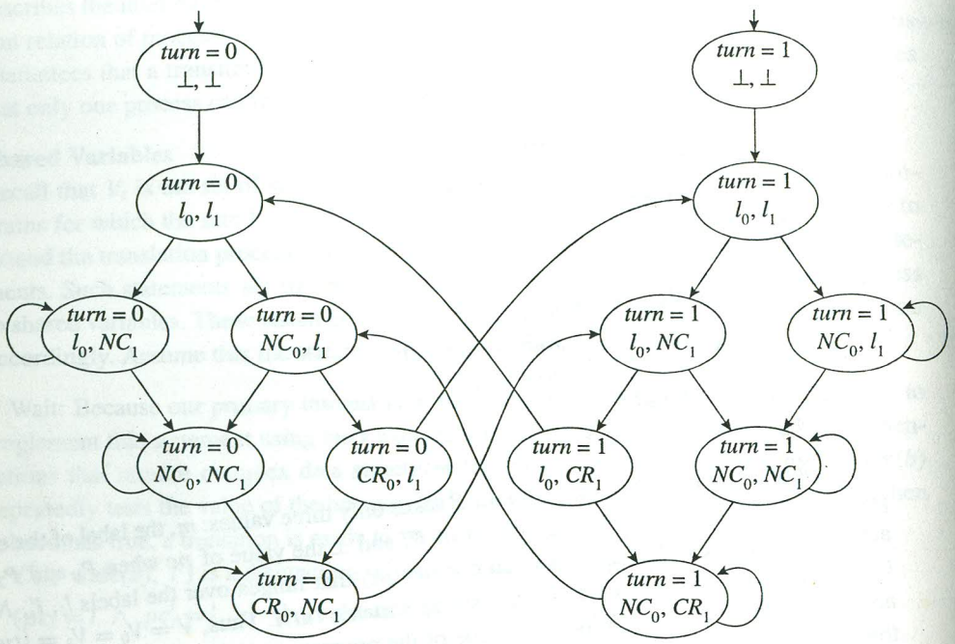


Figure 2.2
Reachable states of Kripke structure for mutual exclusion example.

For each process P_i, $C(l_i, P_i, l_i')$ is the disjunction of:

- $pc_i = l_i \wedge pc_i' = NC_i \wedge True \wedge same(turn)$
- $pc_i = NC_i \wedge pc_i' = CR_i \wedge turn = i \wedge same(turn)$
- $pc_i = CR_i \wedge pc_i' = l_i \wedge turn' = (i+1) \bmod 2$
- $pc_i = NC_i \wedge pc_i' = NC_i \wedge turn \neq i \wedge same(turn)$
- $pc_i = l_i \wedge pc_i' = l_i' \wedge False \wedge same(turn)$

The Kripke structure in Figure 2.2 is derived from the formulas $\mathcal{S}_0$ and $\mathcal{R}$ as described in Section 2.1.1. By examining the state space of the Kripke structure, it is easy to see that the processes will never be in their critical regions at the same time. Thus, the program guarantees the required mutual exclusion property. However, this program fails to guarantee *absence of starvation*, since one of the processes may continuously try to enter its critical region without ever being able to do so, while the other process stays in its critical region forever. Later, we will see how to formulate and model check such properties.

3 Temporal Logics

In this chapter we describe a logic for specifying properties of the state transition systems or Kripke structures introduced in Section 2.1. The logic uses atomic propositions and boolean connectives such as conjunction, disjunction, and negation to build up complicated expressions describing properties of states. In *reactive* systems, we are also interested in describing the transitions between states. This is important because such systems interact with and continually respond to their environment. Traditional software verification methodologies, such as those due to Floyd [114] and Hoare [136], deal with the input-output semantics of programs. The internal details of how the computation is carried out are not reflected in the properties that can be specified and proved; only the input at the start of execution and the output at termination are described. In contrast, for reactive systems, the computation sequence is of primary importance, and many reactive systems are designed not to terminate.

Temporal logic is a formalism for describing sequences of transitions between states in a reactive system. In the temporal logics that we will consider, time is not mentioned explicitly; instead, a formula might specify that *eventually* some designated state is reached, or that an error state is *never* entered. Properties like *eventually* or *never* are specified using special *temporal operators*. These operators can also be combined with boolean connectives or nested arbitrarily. Temporal logics differ in the operators that they provide and the semantics of those operators. We will focus on a powerful logic called CTL* [61, 63, 105].

3.1 The Computation Tree Logic CTL*

Conceptually, CTL* formulas describe properties of *computation trees*. The tree is formed by designating a state in a Kripke structure as the *initial state* and then unwinding the structure into an infinite tree with the designated state at the root, as illustrated in Figure 3.1. The computation tree shows all of the possible executions starting from the initial state.

In CTL* formulas are composed of *path quantifiers* and *temporal operators*. The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers **A** ("for all computation paths") and **E** ("for some computation path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have some property. The temporal operators describe properties of a path through the tree. There are five basic operators:

- **X** ("next time") requires that a property holds in the second state of the path.
- The **F** ("eventually" or "in the future") operator is used to assert that a property will hold at some state on the path.
- **G** ("always" or "globally") specifies that a property holds at every state on the path.

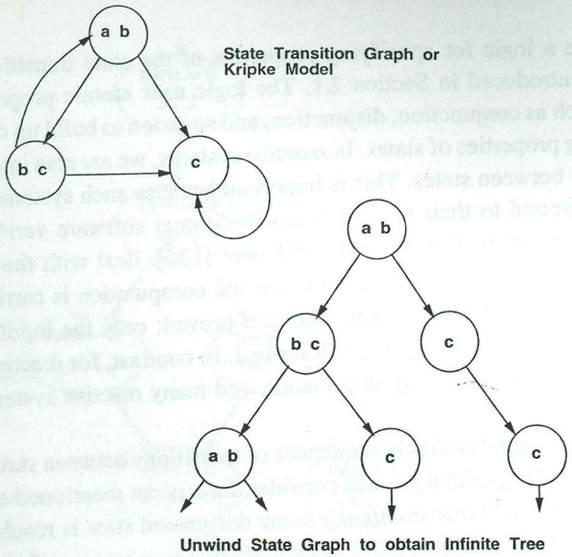


Figure 3.1
Computation trees.

- The **U** ("until") operator is a bit more complicated since it is used to combine two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.
- **R** ("release") is the logical dual of the **U** operator. It requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

The remainder of this section contains a precise description of the syntax and semantics of CTL*. There are two types of formulas in CTL*: *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). Let AP be the set of atomic proposition names. The syntax of state formulas is given by the following rules:

- If $p \in AP$, then p 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 $\mathbf{E}\ f$ and $\mathbf{A}\ f$ are state formulas.

Two additional rules are needed to specify the syntax of path formulas:

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

CTL* is the set of state formulas generated by the above rules.

We define the semantics of CTL* with respect to a Kripke structure. Recall that a Kripke structure M is a triple $\langle S, R, L\rangle$, where S is the set of states; $R \subseteq S \times S$ is the transition relation, which must be *total* (i.e., for all states $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$); and $L : S \to 2^{AP}$ is a function that labels each state with a set of atomic propositions true in that state. A *path in M* is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. (Alternatively, we can think of a path as an infinite branch in the computation tree that corresponds to the Kripke structure.)

We use π^i to denote the *suffix* of π starting at s_i. If f is a state formula, the notation $M, s \models f$ means that f holds at state s in the Kripke structure M. Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along path π in the Kripke structure M. When the Kripke structure M is clear from the context, we will usually omit it. The relation $\models$ is defined inductively as follows (assuming that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

1. $M, s \models p \Leftrightarrow p \in L(s)$.
2. $M, s \models \neg f_1 \Leftrightarrow M, s \not\models f_1$.
3. $M, s \models f_1 \vee f_2 \Leftrightarrow M, s \models f_1$ or $M, s \models f_2$.
4. $M, s \models f_1 \wedge f_2 \Leftrightarrow M, s \models f_1$ and $M, s \models f_2$.
5. $M, s \models \mathbf{E}\ g_1 \Leftrightarrow$ there is a path π from s such that $M, \pi \models g_1$.
6. $M, s \models \mathbf{A}\ g_1 \Leftrightarrow$ for every path π starting from s, $M, \pi \models g_1$.
7. $M, \pi \models f_1 \Leftrightarrow s$ is the first state of π and $M, s \models f_1$.
8. $M, \pi \models \neg g_1 \Leftrightarrow M, \pi \not\models g_1$.
9. $M, \pi \models g_1 \vee g_2 \Leftrightarrow M, \pi \models g_1$ or $M, \pi \models g_2$.
10. $M, \pi \models g_1 \wedge g_2 \Leftrightarrow M, \pi \models g_1$ and $M, \pi \models g_2$.
11. $M, \pi \models \mathbf{X}\, g_1 \Leftrightarrow M, \pi^1 \models g_1$.
12. $M, \pi \models \mathbf{F}\, g_1 \Leftrightarrow$ there exists a $k \geq 0$ such that $M, \pi^k \models g_1$.
13. $M, \pi \models \mathbf{G}\, g_1 \Leftrightarrow$ for all $i \geq 0$, $M, \pi^i \models g_1$.
14. $M, \pi \models g_1\, \mathbf{U}\, g_2 \Leftrightarrow$ there exists a $k \geq 0$ such that $M, \pi^k \models g_2$ and for all $0 \leq j < k$, $M, \pi^j \models g_1$.
15. $M, \pi \models g_1\, \mathbf{R}\, g_2 \Leftrightarrow$ for all $j \geq 0$, if for every $i < j$ $M, \pi^i \not\models g_1$ then $M, \pi^j \models g_2$.

It is easy to see that the operators $\vee$, $\neg$, **X**, **U**, and **E** are sufficient to express any other CTL* formula.

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$
- $f\ \mathbf{R}\ g \equiv \neg(\neg f\ \mathbf{U}\ \neg g)$
- $\mathbf{F}\ f \equiv \mathit{True}\ \mathbf{U}\ f$
- $\mathbf{G}\ f \equiv \neg\,\mathbf{F}\,\neg f$
- $\mathbf{A}(f) \equiv \neg\,\mathbf{E}(\neg f)$

3.2 CTL and LTL

In this section we consider two useful sublogics of CTL*: one is a *branching-time* logic and one is a *linear-time* logic. The distinction between the two is in how they handle branching in the underlying computation tree. In branching-time temporal logic the temporal operators quantify over the paths that are possible from a given state. In linear-time temporal logic, operators are provided for describing events along a single computation path.

Computation Tree Logic (CTL) [19, 61, 104] is a restricted subset of CTL* in which each of the temporal operators **X**, **F**, **G**, **U**, and **R** must be immediately preceded by a path quantifier. More precisely, CTL is the subset of CTL* that is obtained by restricting the syntax of path formulas using the following rule.

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

Linear Temporal Logic (LTL) [217], on the other hand, will consist of formulas that have the form $\mathbf{A}\ f$ where f is a path formula in which the only state subformulas permitted are atomic propositions. More precisely, an LTL path formula is either:

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

It can be shown [59, 105, 166] that the three logics that we have discussed have different expressive powers. For example, there is no CTL formula that is equivalent to the LTL formula $\mathbf{A}(\mathbf{FG}\ p)$. This formula expresses the property that along every path, there is some state from which p will hold forever. Likewise, there is no LTL formula that is equivalent to the CTL formula $\mathbf{AG}(\mathbf{EF}\ p)$. The disjunction of these two formulas $\mathbf{A}(\mathbf{FG}\ p) \vee \mathbf{AG}(\mathbf{EF}\ p)$ is a CTL* formula that is not expressible in either CTL or LTL.

Most of the specifications in this book will be written in the logic CTL. There are ten basic CTL operators:

- **AX** and **EX**,
- **AF** and **EF**,
- **AG** and **EG**
- **AU** and **EU**,
- **AR** and **ER**.

Each of the ten operators can be expressed in terms of three operators **EX**, **EG**, and **EU**:

- $\mathbf{AX}\ f = \neg\,\mathbf{EX}(\neg f)$
- $\mathbf{EF}\ f = \mathbf{E}[\mathit{True}\ \mathbf{U}\ f]$
- $\mathbf{AG}\ f = \neg\,\mathbf{EF}(\neg f)$
- $\mathbf{AF}\ f = \neg\,\mathbf{EG}(\neg f)$
- $\mathbf{A}[f\ \mathbf{U}\ g] \equiv \neg\,\mathbf{E}[\neg g\ \mathbf{U}\ (\neg f \wedge \neg g)] \wedge \neg\,\mathbf{EG}\ \neg g$
- $\mathbf{A}[f\ \mathbf{R}\ g] \equiv \neg\,\mathbf{E}[\neg f\ \mathbf{U}\ \neg g]$
- $\mathbf{E}[f\ \mathbf{R}\ g] \equiv \neg\,\mathbf{A}[\neg f\ \mathbf{U}\ \neg g]$

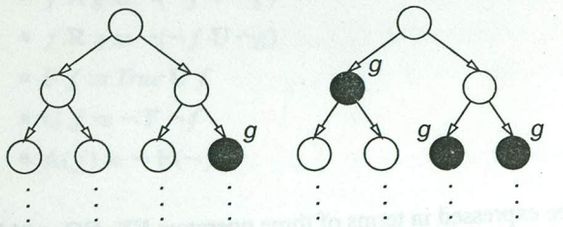
The four operators that are used most widely are illustrated in Figure 3.2. The operators are easiest to understand in terms of the computation tree obtained by unfolding the Kripke model. Each computation tree has the state s_0 as its root.

Some typical CTL formulas that might arise in verifying a finite state concurrent program are given below:

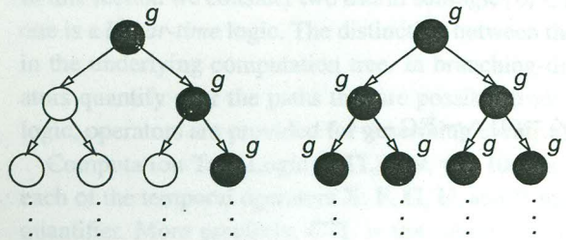
- $\mathbf{EF}(\mathit{Start} \wedge \neg \mathit{Ready})$: It is possible to get to a state where *Start* holds but *Ready* does not hold.
- $\mathbf{AG}(\mathit{Req} \rightarrow \mathbf{AF}\ \mathit{Ack})$: If a request occurs, then it will be eventually acknowledged.
- $\mathbf{AG}(\mathbf{AF}\ \mathit{DeviceEnabled})$: The proposition *DeviceEnabled* holds infinitely often on every computation path.
- $\mathbf{AG}(\mathbf{EF}\ \mathit{Restart})$: From any state it is possible to get to the *Restart* state.

Many of the methods to avoid the state explosion problem rely on compositional reasoning or abstraction. The logic that is typically used in these cases is more restricted and allows only *universal path quantifiers*. The restriction of CTL* to universal path quantifiers is called ACTL*, and the restriction of CTL to universal path quantifiers is called ACTL.

In order to avoid implicit existential path quantifiers resulting from the use of negation, we assume that the formulas are given in *positive normal form*, that is, negations are applied only to atomic propositions. To avoid the loss of expressive power, we need conjunction and disjunction, and both the **U** and **R** operators.



$M, s_0 \models \mathbf{EF}\, g$ $\qquad$ $M, s_0 \models \mathbf{AF}\, g$



$M, s_0 \models \mathbf{EG}\, g$ $\qquad$ $M, s_0 \models \mathbf{AG}\, g$

Figure 3.2
Basic CTL operators.

The formulas **AF AG** a and **AF AX** a are examples of ACTL formulas. These formulas are not expressible in LTL [59]. Because ACTL is a subset of CTL, the logics ACTL and LTL are incomparable. Moreover, ACTL* is more expressive than LTL. The formulas **AG EF** *Start* and **AG** $\neg$ **AF** *Start* are not in ACTL.

3.3 Fairness

Finally, we consider the issue of *fairness*. In many cases, we are only interested in the correctness along fair computation paths. For example, if we are verifying an asynchronous circuit with an arbiter, we may wish to consider only those executions in which the arbiter does not ignore one of its request inputs forever. Alternatively, we may want to consider communication protocols that operate over reliable channels which have the property that no message is ever continuously transmitted but never received. Such properties cannot be expressed directly in CTL [59, 104, 105] but can be expressed in CTL*. In order to deal with fairness in CTL we must modify its semantics slightly. We call the new semantics of the logic the *fair semantics*. A *fairness constraint* can be an arbitrary set of states, usually described by a formula of the logic. If fairness constraints are interpreted as sets of states, then a fair path must contain an element of each fairness constraint infinitely often. If fairness constraints are interpreted as CTL formulas, then a path is *fair* if each constraint is true *infinitely often* along the path. The path quantifiers in the logic are then restricted to fair paths.

Formally, a *fair Kripke structure* is a 4-tuple $M = (S, R, L, F)$, where S, L, and R are defined as before and $F \subseteq 2^S$ is a set of fairness constraints (often called generalized Büchi acceptance conditions). Let $\pi = s_0, s_1, \ldots$ be a path in M. Define

$$\inf(\pi) = \{\, s \mid s = s_i \text{ for infinitely many } i \,\}.$$

We say that π is *fair* if and only if for every $P \in F$, $\inf(\pi) \cap P \neq \emptyset$. The semantics of CTL* with respect to a fair Kripke structure is very similar to the semantics of CTL* with respect to an ordinary Kripke structure. We will write $M, s \models_F f$ to indicate that the state formula f is true in state s of the fair Kripke structure M. Similarly, we write $M, \pi \models_F g$ to indicate that the path formula g is true along path π in M. Only clauses 1, 5 and 6 in the original semantics change.

1. $M, s \models_F p \quad \Leftrightarrow$ there exists a fair path starting from s and $p \in L(s)$.
5. $M, s \models_F \mathbf{E}(g_1) \quad \Leftrightarrow$ there exists a fair path π starting from s such that $\pi \models_F g_1$.
6. $M, s \models_F \mathbf{A}(g_1) \quad \Leftrightarrow$ for all fair paths π starting from s, $\pi \models_F g_1$.

To illustrate the use of fairness, consider again the communication protocol for reliable channels. There is one fairness constraint for each channel that expresses the reliability of that channel. A possible choice for the fairness constraint associated with channel i is the set of states that satisfy the formula $\neg send_i \vee receive_i$. Thus, a computation path is fair if and only if for every channel, infinitely often either a message is not sent or a message is received. Other notions of fairness are dealt with in [116].