

CMPSC 267 Class Notes
Introduction to Temporal Logic and Model Checking

Tevfik Bultan
Department of Computer Science
University of California, Santa Barbara

Chapter 1

Introduction

Static analysis techniques uncover properties of computer systems with the goal of improving them. A system can be improved in two ways using static analysis (1) improving its performance and (2) improving its correctness by eliminating bugs. As an example, consider the static analysis techniques used in compilers. Data flow analysis finds the relationships between the definitions and the uses of variables, then this information is used to generate more efficient executable code to improve the performance. On the other hand, type checking determines the types of the expressions in the source code, which is used to find type errors.

Model checking is a static analysis technique used for improving the *correctness* of computer systems. I refer to model checking as a *static analysis* technique because it is a technique used to find errors before run-time. For example, run-time checking of assertions is not a static analysis technique, it only reports an assertion failure at run-time, after it occurs. In model checking the goal is more ambitious, we try to verify that the assertions hold on every possible execution of a system (or program or protocol or whatever you call the thing you are verifying). Given its ambitious nature, model checking is computationally expensive. (Actually, it is an uncomputable problem for most types of infinite state systems.)

Model checkers have been used in checking correctness of high level specifications of both hardware [CK96] and software [CAB⁺98] systems. The motivation is to provide designers with easy and fast feedback, so that the errors can be detected as early as possible. More recently, researchers have developed model checkers for programming languages such as Java and C. Such tools are targeted towards the error-prone parts of programs such as concurrent programming [HP00, BHPV00, CBH⁺00] or systems programming (for example, debugging device drivers for PCs) [BR01]. These program checkers rely heavily on state-space reduction techniques and abstractions [CBH⁺00, BMMR01, BPR01].

1.1 Model Checking: A Brief History

In the late 70's, Pnueli observed that the notions such as termination, partial correctness and total correctness, which dominated the earlier work in automated verification, were not suitable for characterizing properties of *reactive systems* [Pnu77, Pnu81]. Reactive systems (as opposed to transformational systems) are systems which keep interacting with their environment without terminating. Examples of reactive systems are protocols, concurrent programs, operating systems, and hardware systems. In contrast, transformational systems receive an input, do some computation and output a result. Earlier work on verification concentrated on verification of transformational systems.

To put the Pnueli's work in context, I will briefly discuss what partial correctness and total correctness mean. Assume that we are given a program (or program segment) S and assertions P and Q . In partial correctness, the goal is the following: prove that if the program S starts in a state satisfying the assertion P and if the execution terminates, then the final state satisfies the assertion Q . In total correctness, however, the goal is to prove that the final states satisfy Q *and* the execution terminates [Gri81]. Work on verification of transformational systems builds on earlier work on axiomatic verification by Floyd [Flo67] and Hoare [Hoa69], and Dijkstra's work on weakest preconditions [Dij75, Dij76]. Hoare introduced the Hoare triple notation $P\{S\}Q$ to state the correctness assumptions about a program segment S [Hoa69]. Hoare triple $P\{S\}Q$ indicates the partial correctness condition explained above. Similarly, $\{P\}S\{Q\}$ is used to indicate the total correctness condition [Gri81]. Dijkstra's weakest precondition operator (indicated as wp) is defined as follows: weakest precondition of a program (or program segment) S with respect to an assertion Q would be the set of program states such that when S begins executing in such a state then it is guaranteed to terminate in a state satisfying Q [Dij75, Dij76, Gri81]. For example, $wp(i:=i+4, i \geq 6) \equiv i \geq 2$. Note that, $\{P\}S\{Q\}$ is equivalent to $P \Rightarrow wp(S, Q)$.

Earlier work on verification concentrated on developing proofs of correctness by using the above concepts. Note that, one can develop such a proof incrementally by propagating the weakest preconditions from program statement to program statement. One gets in to trouble in the presence of loops (or recursive procedures) but these can be handled if one can come up with the appropriate *loop invariant*, i.e., an assertion which holds during every iteration of the loop and is strong enough to generate a useful weakest precondition. A lot of effort was spent on educating computer scientists so that they can develop proofs of correctness while they are developing their programs. Given the tedious nature of such a task, automating the verification task seems like a good idea. However uncomputability results imply that most verification tasks cannot be automated completely. Earlier work on automated verification focused on automating the weakest precondition computations and semi-automatically generating proofs in Hoare

Logics using interactive theorem provers where user may supply a loop invariant that the automated theorem prover is not able to generate.

As I stated above, Pnueli observed that for verification of reactive systems a different approach was necessary. First of all, the notion of termination is irrelevant to reactive systems. We are not interested what happens when a reactive system terminates (in fact we do not expect a reactive system to terminate). We are interested in what is happening when the system is executing. The reason Pnueli's work ended up being very influential is that most systems for which correctness is crucial are reactive systems. Typical examples are protocols, concurrent programs, operating systems, and hardware systems. Properties of reactive systems involve notions of *invariance* (an assertion always holds in every state of the execution), *eventuality* (an assertion eventually holds in some state in every execution of the system), and *fairness* (if a process or a thread is enabled, it should eventually be scheduled for execution). Temporal logics (which are based on modal logics) were introduced by Pnueli as a formalism for characterizing such properties [Pnu77, Pnu81]. It is possible to express properties such as invariance, eventuality and fairness, in temporal logics using simple temporal operators.

1.1.1 An Example

Let's see a small example to differentiate the transformational and reactive approaches to verification. Assume that we are given two procedures:

```
produce() {
  if (numItems < size)
    numItems := numItems + 1;
}
```

```
consume() {
  if (numItems > 0)
    numItems := numItems - 1;
}
```

Now, we can look at these procedures as transformational systems (where `numItems` and `size` are global variables serving as input and output) and we may want to prove that these procedures preserve the property $0 \leq \text{numItems} \leq \text{size}$. How can we do this? Well, this translates to proving

$$0 \leq \text{numItems} \leq \text{size} \{ \text{produce} \} 0 \leq \text{numItems} \leq \text{size}$$

and

$$0 \leq \text{numItems} \leq \text{size} \{ \text{consume} \} 0 \leq \text{numItems} \leq \text{size}.$$

4

Which are equivalent to

$$0 \leq \text{numItems} \leq \text{size} \Rightarrow \text{wp}(\text{produce}, 0 \leq \text{numItems} \leq \text{size})$$

and

$$0 \leq \text{numItems} \leq \text{size} \Rightarrow \text{wp}(\text{consume}, 0 \leq \text{numItems} \leq \text{size}).$$

Ignoring the procedure call and the procedure return for simplicity, what we really want to prove is:

$$\begin{aligned} &0 \leq \text{numItems} \leq \text{size} \Rightarrow \\ &\text{wp}(\text{if } (\text{numItems} < \text{size}) \text{ numItems} := \text{numItems} + 1, \\ &0 \leq \text{numItems} \leq \text{size}) \end{aligned}$$

and

$$\begin{aligned} &0 \leq \text{numItems} \leq \text{size} \Rightarrow \\ &\text{wp}(\text{if } (\text{numItems} > 0) \text{ numItems} := \text{numItems} - 1, \\ &0 \leq \text{numItems} \leq \text{size}). \end{aligned}$$

So, how do we prove these? Actually, one can show that (using the semantics of the if statement):

$$\text{wp}(\text{if } B \text{ } S, Q) \equiv (B \Rightarrow \text{wp}(S, Q)) \wedge (\neg B \Rightarrow Q)$$

where B is a boolean condition. If we instantiate this rule for the above examples we will get:

$$\begin{aligned} &\text{wp}(\text{if } (\text{numItems} < \text{size}) \text{ numItems} := \text{numItems} + 1, \\ &0 \leq \text{numItems} \leq \text{size}) \\ &\equiv (\text{numItems} < \text{size} \Rightarrow \text{wp}(\text{numItems} := \text{numItems} + 1, \\ &0 \leq \text{numItems} \leq \text{size})) \\ &\wedge (\text{numItems} \geq \text{size} \Rightarrow 0 \leq \text{numItems} \leq \text{size}) \end{aligned}$$

and

$$\begin{aligned} &\text{wp}(\text{if } (\text{numItems} > 0) \text{ numItems} := \text{numItems} - 1, \\ &0 \leq \text{numItems} \leq \text{size}) \\ &\equiv (\text{numItems} > 0 \Rightarrow \text{wp}(\text{numItems} := \text{numItems} - 1, \\ &0 \leq \text{numItems} \leq \text{size})) \\ &\wedge (\text{numItems} \leq 0 \Rightarrow 0 \leq \text{numItems} \leq \text{size}) \end{aligned}$$

OK, now we need a rule for dealing with weakest precondition of assignments which simply is:

$$\text{wp}(x := e, Q) \equiv Q[x \leftarrow e],$$

where, $Q[x \leftarrow e]$ means substituting e in place of x in Q . If we apply this rule to the above example we get:

$$\begin{aligned} &\text{wp}(\text{numItems} := \text{numItems} + 1, 0 \leq \text{numItems} \leq \text{size}) \\ &\equiv 0 \leq \text{numItems} + 1 \leq \text{size}) \end{aligned}$$

and

$$\begin{aligned} & wp(\text{numItems} := \text{numItems} - 1, 0 \leq \text{numItems} \leq \text{size}) \\ & \equiv 0 \leq \text{numItems} - 1 \leq \text{size}. \end{aligned}$$

Now combining the above steps we get:

$$\begin{aligned} & 0 \leq \text{numItems} \leq \text{size} \{\text{produce}\} 0 \leq \text{numItems} \leq \text{size} \\ & \equiv 0 \leq \text{numItems} \leq \text{size} \Rightarrow \\ & ((\text{numItems} < \text{size} \Rightarrow (0 \leq \text{numItems} + 1 \leq \text{size})) \\ & \quad \wedge (\text{numItems} \geq \text{size} \Rightarrow (0 \leq \text{numItems} \leq \text{size}))) \\ & \equiv \text{True} \end{aligned}$$

and

$$\begin{aligned} & 0 \leq \text{numItems} \leq \text{size} \{\text{consume}\} 0 \leq \text{numItems} \leq \text{size} \\ & \equiv 0 \leq \text{numItems} \leq \text{size} \Rightarrow \\ & ((\text{numItems} > 0 \Rightarrow (0 \leq \text{numItems} - 1 \leq \text{size})) \\ & \quad \wedge (\text{numItems} \leq 0 \Rightarrow (0 \leq \text{numItems} \leq \text{size}))) \\ & \equiv \text{True}. \end{aligned}$$

With some arithmetic manipulation we can show that the formulas above are equivalent to *True*. Hence we achieved our objective and show that the program segments **produce** and **consume** preserve the property $0 \leq \text{numItems} \leq \text{size}$.

The above proof was using the transformational view and axiomatic reasoning. One observation is that such proofs are difficult to do by hand. Even for the small example we discussed above, the proof requires manipulation of long arithmetic formulas. Another observation is that based on the above proof we only now that if the property $0 \leq \text{numItems} \leq \text{size}$ holds at the entry to the procedures **produce** and **consume**, it also holds at the exit. Here are the issues we would like to address:

- What about if we wanted to prove that $0 \leq \text{numItems} \leq \text{size}$ was an *invariant*, i.e., it holds at each step of the execution? Or we may want to prove that a property will hold *eventually*.
- Another complication arises when we want to prove the above property for a reactive system where procedures **produce** and **consume** are called by concurrent processes. Actually, (without atomicity assumptions) one can easily show that the invariant $0 \leq \text{numItems} \leq \text{size}$ can be violated if two concurrent processes execute **produce** and **consume** procedures concurrently. To preserve the invariant one needs to use synchronization among concurrent processes and some atomicity assumptions.
- Another issue is the following question: Can we automate the above proof and how?

Model checking research extends the earlier work on verification by addressing the issues I listed above. To summarize, 1) model checking concentrates on reactive systems, 2) it deals with verification of *temporal* properties, and 3) it is algorithmic, it focuses on automated verification algorithms rather than proof by hand.

1.1.2 Model Checking Research

Soon after the introduction of temporal logics it was observed that, for finite transition systems, temporal properties can be verified automatically [CE81, QS82]. A procedure which checks if a transition system is a model for a temporal logic formula is called a *model checker*. Generally, this is achieved using state exploration—searching the state space of a system exhaustively to verify (or falsify) a property [CES86].

Automation of the verification task is crucial to model checking. Verification using automated theorem provers is generally not completely automated (this is not surprising, since most of the time the applications are uncomputable problems). The user needs to interact with the theorem prover when the theorem prover gets stuck, and for example provide the loop invariant. In model checking the idea is to verify a system completely automatically without any human intervention. We will see that this is not always the case (but at least it is the motivation).

Another difference between the model checking and the theorem proving approaches is the fact that model checkers are mostly used for debugging, i.e., to falsify a property by generating a counter-example execution that demonstrates that the stated property is incorrect. This is one of the reasons that model checking approach has become popular.

There are two different approaches to model checking. One of them is using symbolic search techniques [BCM⁺90, CBM89, CMB90, McM93]. The main idea is to represent sets of states and transition relations symbolically (implicitly), rather than using an explicit representation that uses a fixed amount of storage per state or transition. Symbolic representations significantly increased the sizes of the transition systems that can be analyzed [BCM⁺90]. Boolean formulas is the first (and the most common) symbolic representation used in model checking. Symbolic state exploration techniques which use this encoding rely heavily on efficient data structures for manipulating Boolean formulas. *Binary decision diagrams* (BDDs) [Bry86] have proved to be suitable for this task [BCM⁺90, McM93]. BDDs support all the functionality required in model checking computations [Bry86]. SMV is a BDD-based symbolic model checker [McM93] which has been successfully used in various case studies.

Work on symbolic model checking lead to infinite state verification tools which can verify infinite state systems (for example real-time and hybrid systems [ACH⁺95, AHH96] and concurrent programs with unbounded integer variables [BGP97, BGP99, DP99, DP01]). Verification problems for such systems is generally uncomputable. How-

ever, one can use heuristics and semi-procedures to conservatively verify such systems. At a certain level infinite state model checking and verification based on automated theorem provers are similar. One difference is that the infinite state model checkers tend to be more domain specific and therefore able to exploit domain specific heuristics.

The second approach to model checking is automata-based [VW86]. It involves converting the negation of the given temporal property to an equivalent property automaton. Every (infinite) word accepted by the property automaton corresponds to a behavior that satisfies the negated property. The behavior of this automaton is compared with the behavior of the automaton that represents the system by generating a product automaton. If the product automaton is empty, then the system satisfies the original property.

The efficiency of automata-based model checking can be improved using partial-order reduction techniques [God94]. SPIN is a finite state model checker which uses automata-based approach and has been successfully used in protocol verification [Hol97].

Recently, both symbolic and automata-based model checking technique resulted in verification tools for verifying C and Java programs [HP00, BHPV00, God97b, God97a, BR01]. These tools rely on application of various abstraction techniques to programs to construct a reduced state space that can be automatically analyzed [CBH⁺00, BMMR01, CC77, GS97].

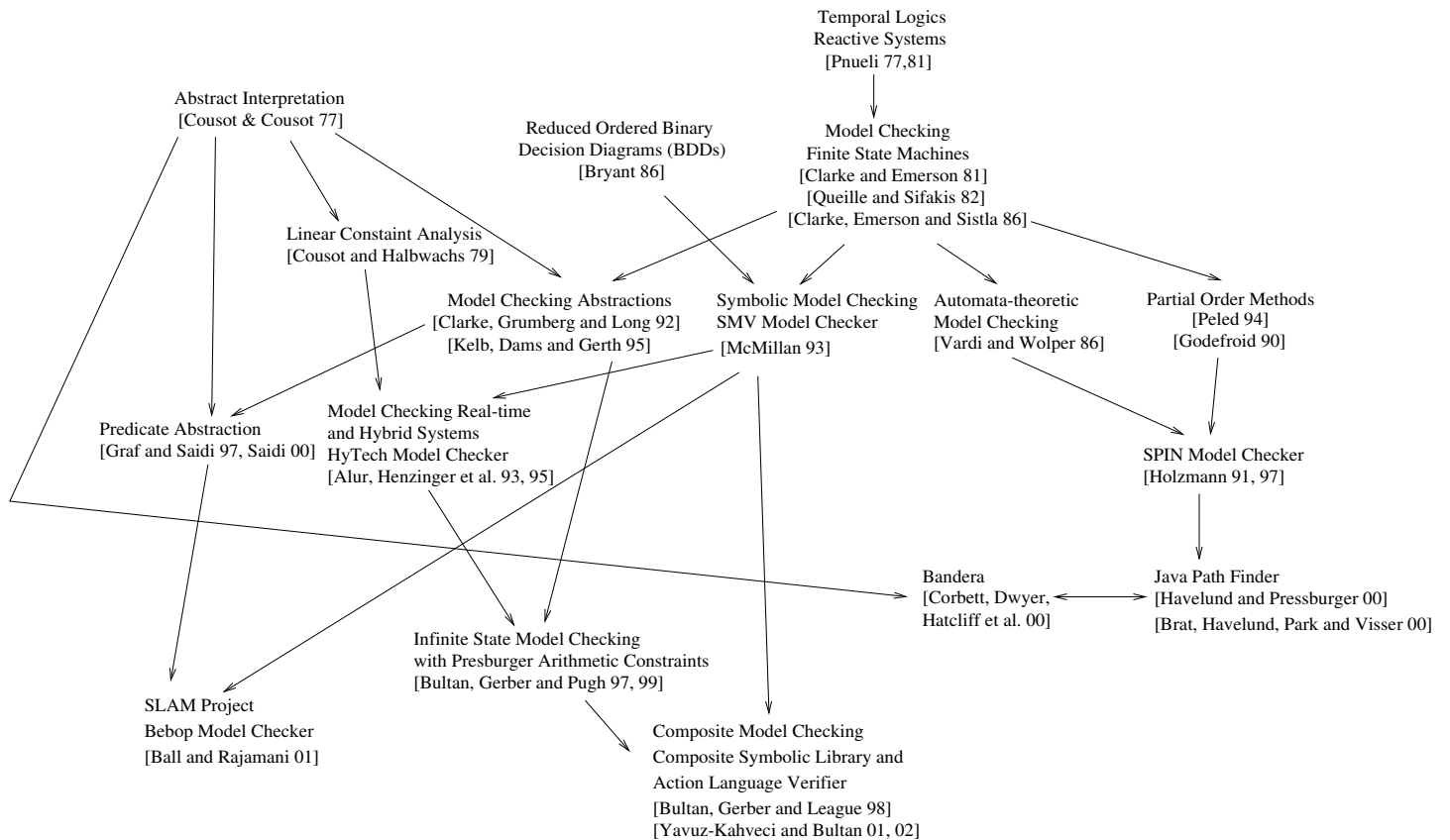


Figure 1.1: A genealogy of model checking tools. Note that there are lots of interesting papers which are not included here. This is just a very rough summary of the results that relate to the tools we will use in this course.

Chapter 2

Transition Systems: A Mathematical Model for Reactive Systems

2.1 Specifying a system with logical formulas

We will use logical formulas to specify a reactive system. We will consider that a system $P = (V, IC, RF)$ is represented by (1) a finite *set of variables* V ; (2) an *initial condition* IC , which specifies the starting states of the system; and (3) a transition relation formula RF .

To represent the transition relation formula we will use variable names in V to denote the current state value of the variables and primed variable names in V' to denote the next state value of the variables, i.e., RF will be a formula on variables in V and V' . For example, consider the procedure:

```
produce() {  
a1: if (numItems < size)  
a2:   numItems := numItems + 1;  
a3: }
```

For this procedure $V = \{pc, numItems, size\}$ where pc is the variable that corresponds to the program counter. The labels $\{a1, a2, a3\}$ denote the program points, i.e., they are the values that the program counter can take. Then, the transition relation formula RF is defined on the sets $V = \{pc, numItems, size\}$ and $V' = \{pc', numItems', size'\}$

$$\begin{array}{l}
V = \{a, b, turn, pc_1, pc_2\} \\
a, b : \text{boolean} \\
turn : \{0, 1\} \\
pc_1, pc_2 : \{T, W, C\} \\
\\
IC \equiv \neg a \wedge \neg b \wedge pc_1 = T \wedge pc_2 = T \\
\\
RF \equiv pc_1 = T \wedge a' \wedge turn' = 1 \wedge pc_1' = W \wedge \text{SAME}(\{pc_2, b\}) \\
\vee pc_1 = W \wedge (\neg b \vee turn = 0) \wedge pc_1' = C \wedge \text{SAME}(\{pc_2, a, b, turn\}) \\
\vee pc_1 = C \wedge pc_1' = T \wedge \neg a' \wedge \text{SAME}(\{pc_2, b, turn\}) \\
\vee pc_2 = T \wedge b' \wedge turn' = 0 \wedge pc_2' = W \wedge \text{SAME}(\{pc_1, a\}) \\
\vee pc_2 = W \wedge (\neg a \vee turn = 1) \wedge pc_2' = C \wedge \text{SAME}(\{pc_2, a, b, turn\}) \\
\vee pc_2 = C \wedge pc_2' = T \wedge \neg b' \wedge \text{SAME}(\{pc_1, a, turn\})
\end{array}$$

Figure 2.1: An finite state mutual exclusion algorithm

as follows:

$$\begin{array}{l}
RF \equiv pc = a1 \wedge (numItems < size) \wedge pc' = a2 \wedge \text{SAME}(\{numItems, size\}) \\
\vee pc = a1 \wedge (\neg(numItems < size)) \wedge pc' = a3 \wedge \text{SAME}(\{numItems, size\}) \\
\vee pc = a2 \wedge numItems' = numItems + 1 \wedge pc' = a3 \wedge \text{SAME}(\{size\})
\end{array}$$

We use $\text{SAME}(U)$ where $U \subseteq V$ to denote the conjunction

$$\text{SAME}(U) \equiv \bigwedge_{v \in U} v' = v,$$

i.e., $\text{SAME}(U)$ means that the variables in the set U preserve their value. Note that, in the formula RF the primed identifiers correspond to the next state values, and the unprimed identifiers correspond to the current state values of the variables. Hence, the symbol v' denotes the new value of v after a single step of execution.

As another example consider a mutual exclusion algorithm for two processes given in Figure 2.1 (This is a simplified version of Dekker's mutual exclusion algorithm for two processes). This algorithm ensures mutual-exclusion between two processes in accessing a critical section (program counter values mean T : thinking, W : waiting to enter the critical section, and, C : in critical section). You can think of the transition relation formula in Figure 2.1 corresponding to the semantics of a program written in some programming language. It would be interesting to know if this algorithm satisfies both *mutual-exclusion* (two processes never access the critical section at the same time) and *starvation-freedom* (if a process wants to enter the critical section it eventually gets in).

Note that, in the first disjunct in the transition formula

$$pc_1 = T \wedge a' \wedge turn' = 1 \wedge pc_1' = W \wedge \text{SAME}(\{pc_2, b\})$$

$$\begin{array}{l}
V = \{a, b, pc_1, pc_2\} \\
a, b : \text{nonnegative integer} \\
pc_1, pc_2 : \{T, W, C\} \\
\\
IC \equiv a = b = 0 \wedge pc_1 = T \wedge pc_2 = T \\
\\
RF \equiv pc_1 = T \wedge pc'_1 = W \wedge a' = b + 1 \wedge \text{SAME}(\{pc_2, b\}) \\
\vee pc_1 = W \wedge pc'_1 = C \wedge (a < b \vee b = 0) \wedge \text{SAME}(\{pc_2, a, b\}) \\
\vee pc_1 = C \wedge pc'_1 = T \wedge a' = 0 \wedge \text{SAME}(\{pc_2, b\}) \\
\vee pc_2 = T \wedge pc'_2 = W \wedge b' = a + 1 \wedge \text{SAME}(\{pc_1, a\}) \\
\vee pc_2 = W \wedge pc'_2 = C \wedge (b < a \vee a = 0) \wedge \text{SAME}(\{pc_1, a, b\}) \\
\vee pc_2 = C \wedge pc'_2 = T \wedge b' = 0 \wedge \text{SAME}(\{pc_1, a\})
\end{array}$$

Figure 2.2: An infinite state mutual exclusion algorithm

variables a and $turn$ are updated simultaneously. This kind of atomicity could be too coarse-grain in general. If we had to model the case that only one update can be done in one execution step, we could have introduced another program point T_1 and split the above step to two steps:

$$\begin{array}{l}
pc_1 = T \wedge a' \wedge pc'_1 = T_1 \wedge \text{SAME}(\{pc_2, b, turn\}) \\
\vee pc_1 = T_1 \wedge turn' = 1 \wedge pc'_1 = W \wedge \text{SAME}(\{pc_2, b, a\})
\end{array}$$

Of course the resulting transition relation formulas are different. This kind of logical formulation of the transition relation makes the atomicity assumptions (and almost all the assumptions about the semantics transparent).

Another mutual exclusion algorithm [?] for two processes is given in Figure 2.2 (This is a simplified, coarse grain version of the Bakery algorithm for two processes [?]). Note that this algorithm is infinite state since it is using integer variables which are not bounded. In this algorithm, when a process wants to enter the critical section, it first gets a ticket, which will be higher than those of all other processes currently in the critical section or waiting for entry. In the above system, variables a and b hold the ticket values for processes 1 and 2, respectively; a process gets its ticket by simply adding one to the highest outstanding ticket number. Note that variables a and b can increase without bound, i.e., this is not a finite-state program. Again we would like to know if this algorithm satisfies mutual-exclusion and starvation-freedom properties. In the following sections we show how these properties can be formalized using temporal logics.

2.2 From formulas to transition systems

Transition systems are simple, general mathematical models for computer systems. A *transition system* $T = (S, I, R)$ consists of a *set of states* S , a set of *initial states* $I \subseteq S$, and a *transition relation* $R \subseteq S \times S$. Note that, our logical representation of a system $P = (V, IC, RF)$ and the transition system $T = (S, I, R)$ are not far apart. We define the set of states S as the Cartesian product of the domains of the variables in V . Each state $s \in S$ corresponds to a valuation of all the variables in V , i.e., if you assign a value to each variable in V you obtain a state. Then, given a program with n variables $V = \{v_1, v_2, \dots, v_n\}$, the state space is $S \equiv \text{DOMAIN}(v_1) \times \text{DOMAIN}(v_2) \times \dots \times \text{DOMAIN}(v_n)$ where $\text{DOMAIN}(v_i)$ denotes the domain of variable v_i . Each state $s \in S$ corresponds to a valuation of all the variables of the program

$$s \equiv \bigwedge_{i=1}^n v_i = x_i$$

where $x_i \in \text{DOMAIN}(v_i)$. The set of initial states will be the states which satisfy the initial condition IC , i.e., $s \in I$ if and only if $s \models IC$. Similarly, the transition relation is defined as the set of state pairs (s, s') such that the valuations corresponding to s and s' satisfy the formula RF , i.e., $(s, s') \in R$ if and only if $(s, s') \models RF$.

For example, for the algorithm given in Figure 2.2, a possible state of the system is:

$$s \equiv pc_1 = T \wedge pc_2 = C \wedge a = 0 \wedge b = 1$$

This is a state where process one is thinking and process two is in the critical section. For this algorithm the set of initial states has only one state in it, which is:

$$s \equiv pc_1 = T \wedge pc_2 = T \wedge a = 0 \wedge b = 0$$

Figure 2.3 shows a part of the infinite transition system for this algorithm.

The algorithm given in Figure 2.1 has a finite state space and its transition system is shown in Figure 2.4.

Generally, in model checking literature, the transition systems are restricted in two ways:

- 1) The transition systems are restricted to finite transition systems (i.e., S is assumed to be finite which is not the case for the Bakery algorithm).
- 2) The transition relation R is assumed to be total, i.e., for each state $s \in S$ there exists a next state s' such that $(s, s') \in R$.

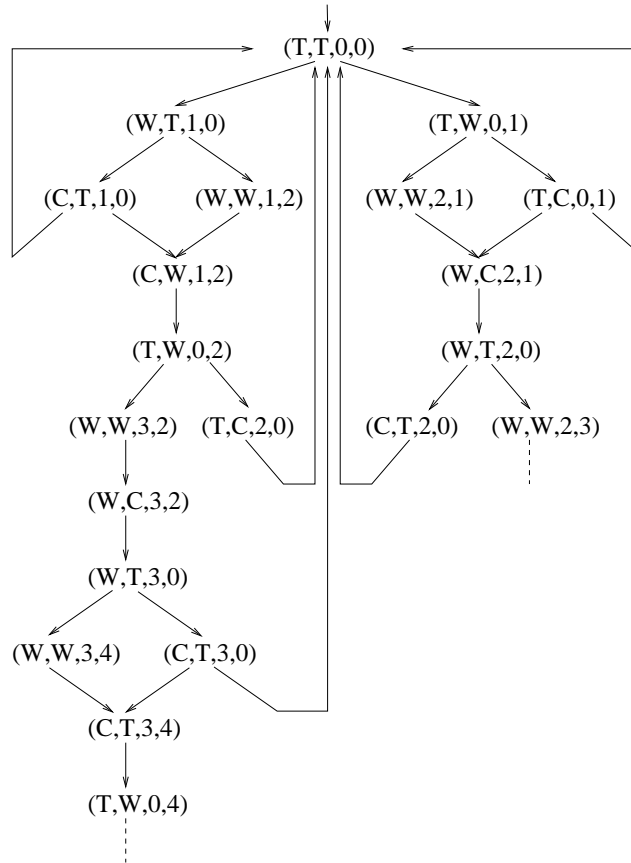


Figure 2.3: Part of the infinite transition system of the algorithm given in Figure 2.2

In the definition of temporal logics in the following sections we will assume that these restrictions hold. However, it is possible to generalize the definitions and remove these restrictions.

Temporal properties of a transition system are specified on its paths. We will define a *path* (s_0, s_1, s_2, \dots) in a transition system $T = (S, I, R)$, as an infinite sequence of states such that for each successive pair of states $(s_i, s_{i+1}) \in R$. We call s' a *successor* of state s if $(s, s') \in R$. Similarly, s' is called a *predecessor* of state s if $(s', s) \in R$. Given a path $x = (s_0, s_1, s_2, \dots)$, x^i denotes the i 'th suffix of the path x , i.e., $x^i = (s_i, s_{i+1}, s_{i+2}, \dots)$, and x_i denotes the i 'th state on x , i.e., $x_i = s_i$. An *execution path* is a path starting from an initial state, i.e., (s_0, s_1, s_2, \dots) is an execution path if $s_0 \in I$.

Valuations of the variables (pc1,pc2,a,b,turn) define the states

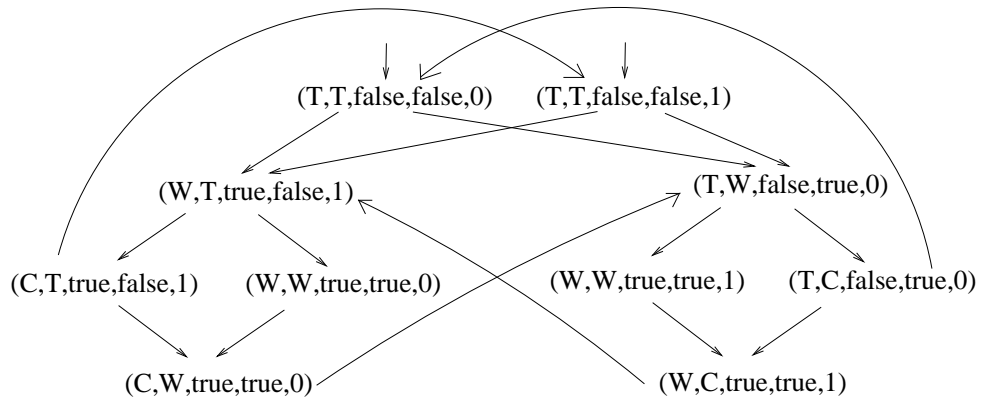


Figure 2.4: Transition system of the algorithm given in Figure 2.1

Chapter 3

Specifying Properties of Reactive Systems: Temporal Logics

Temporal logics are based on modal logics [Eme90, Pnu77, Pnu81]. Modal logics were originally proposed to study situations where truth of a statement depends on its mode. For example, an assertion p may be false *now*, but given a mode *future*, *future* p could be true, i.e., p may become true in the future. Pnueli recognized that such logics were suitable formalisms for specifying properties of reactive systems [Pnu77, Pnu81]. This view has been confirmed by their extensive use in the last two decades.

Temporal logics are composed of a nontemporal part for specifying basic properties of states, plus a set of temporal operators for specifying temporal properties. Nontemporal properties can be checked on each individual state by observing the values of the variables in that state, whereas verifying temporal properties involves investigating paths of the system.

First, we discuss nontemporal portion of temporal logics, i.e., specification of basic properties of states. We will call such properties *atomic properties*. Given a transition system $T = (S, I, R)$, we define AP to be the set of atomic properties of the system. We assume that we have a procedure $L : S \times AP \rightarrow \{\mathbf{true}, \mathbf{false}\}$ such that given a property $p \in AP$ and a state $s \in S$, procedure L can decide if $s \models p$, i.e., $s \models p$ iff $L(s, p) = \mathbf{true}$.

In propositional temporal logics AP consists of propositional logic formulas, whereas in a first order temporal logic it can include formulas with functions, predicates and quantification. In the discussions below, we do not define the set AP . We just state that there exists a decision procedure L which can determine the truth value of the formulas in AP .

3.1 Temporal logics CTL, LTL and CTL*

Various temporal logics have been defined in the last two decades. We will discuss LTL, CTL and CTL* since they are the most commonly used temporal logics. All these logics use four temporal operators G, F, U, X with the following intuitive meanings:

- G*p* Henceforth *p* (or Globally *p*), i.e., assertion *p* holds in every state.
- F*p* Eventually *p* (or Future *p*), i.e., assertion *p* will hold in a future state.
- p*U*q* *p* Until *q*, i.e., assertion *p* will hold until *q* holds.
- X*p* Next *p*, i.e., assertion *p* will hold in the next state.

3.1.1 LTL

In LTL (Linear time Temporal Logic) truth of temporal logic formulas are defined on paths. This supports the linear time view where computation starting from a state is seen as a single sequence of states, i.e., a path. LTL uses the four basic temporal operators G, F, U, X mentioned above.

The following grammar defines the syntax of LTL, where terminals \neg , \wedge , and \vee are logical connectives; terminals X, and U are temporal operators; terminal *ap* denotes an atomic property, (i.e., $ap \in AP$); and nonterminal *f* denotes an LTL formula:

$$f ::= ap \mid f \wedge f \mid f \vee f \mid \neg f \mid X f \mid f U f$$

We have the following equivalences for LTL operators,

$$F p \equiv \mathbf{true} U p \qquad G p \equiv \neg(\mathbf{true} U \neg p)$$

Based on these equivalences, the set $\{\neg, \vee, X, U\} \cup AP$ forms a basis for LTL, i.e., we can express any LTL property using only symbols from this set.

In Table 3.1 we define the semantics of LTL. Based on the rules given in Table 3.1, we can decide if a path in a transition system satisfies an LTL formula.

We also define the truth values of LTL formulas for transition systems:

Given a transition system $T = (S, I, R)$, and a temporal formula *p*, $T \models p$ if for all states $s \in I$, for all paths x such that $x_0 = s$, $x \models p$.

Hence, a transition system satisfies an LTL formula if all the paths starting from its initial states (i.e., all the execution paths) satisfy the formula.

$x \models p$	iff	$L(x_0, p) = \mathbf{true}$ where $p \in AP$.
$x \models \neg p$	iff	not $x \models p$.
$x \models p \wedge q$	iff	$x \models p$ and $x \models q$.
$x \models p \vee q$	iff	$x \models p$ or $x \models q$.
$x \models X p$	iff	$x^1 \models p$.
$x \models p U q$	iff	there exists an i such that $x^i \models q$, and for all $j < i$, $x^j \models p$.
$x \models Fp$	iff	there exists an i such that $x^i \models p$.
$x \models Gp$	iff	for all i , $x^i \models p$.

Table 3.1: LTL Semantics

3.1.2 CTL

CTL (Computation Tree Logic) is a branching time logic, i.e., the computation starting from a state is viewed as a tree where each branch corresponds to a path. In CTL truth of a temporal formula is defined on states. A temporal formula is true for a state if all the paths or some of the paths originating from that state satisfy a condition.

CTL temporal operators are generated from the basic temporal operators G, F, U, X by adding one of the path quantifiers A or E as a prefix. Path quantifier A means “for all paths”, whereas E means “there exists a path”.

The following grammar defines the syntax of CTL, where terminals \neg , \wedge , and \vee are logical connectives; terminals EX, AX, EU and AU are temporal operators; terminal ap denotes an atomic property, (i.e., $ap \in AP$); and nonterminal f denotes a CTL formula:

$$f ::= ap \mid f \wedge f \mid f \vee f \mid \neg f \mid EX f \mid AX f \mid f EU f \mid f AU f$$

We have the following equivalences for CTL operators,

$$\begin{aligned} EF p &\equiv \mathbf{true} EU p & EG p &\equiv \neg(\mathbf{true} AU \neg p) \\ AF p &\equiv \mathbf{true} AU p & AG p &\equiv \neg(\mathbf{true} EU \neg p) \\ AX p &\equiv \neg EX \neg p \end{aligned}$$

Based on these equivalences, the set $\{\neg, \vee, EX, EU, AU\} \cup AP$ forms a basis for CTL.

In Table 3.2 we define the semantics of CTL. Based on the rules given in Table 3.2, we can decide if a state in a transition system satisfies a CTL formula. Semantics of the temporal operators EF, AF, EG and AG follow from the equivalences given above and the rules given in Table 3.2.

We also define the truth value of CTL formulas for transition systems:

Given a transition system $T = (S, I, R)$, and a temporal formula p , $T \models p$ if for all states $s \in I$, $s \models p$.

$s \models p$	iff	$L(s, p) = \mathbf{true}$ where $p \in AP$.
$s \models \neg p$	iff	not $s \models p$.
$s \models p \wedge q$	iff	$s \models p$ and $s \models q$.
$s \models p \vee q$	iff	$s \models p$ or $s \models q$.
$s_0 \models \text{EX } p$	iff	there exists a path (s_0, s_1, s_2, \dots) , such that, $s_1 \models p$.
$s_0 \models \text{AX } p$	iff	for all paths (s_0, s_1, s_2, \dots) , $s_1 \models p$.
$s_0 \models p \text{ EU } q$	iff	there exists a path (s_0, s_1, s_2, \dots) , such that, there exists an i , $s_i \models q$, and for all $j < i$, $s_j \models p$.
$s_0 \models p \text{ AU } q$	iff	for all paths (s_0, s_1, s_2, \dots) , there exists an i , $s_i \models q$, and for all $j < i$, $s_j \models p$.

Table 3.2: CTL Semantics

Hence, a transition system satisfies a CTL formula if all its initial states satisfy the formula.

In Figure 3.1, we show the truth values of temporal properties $\text{EF}p$, $\text{AF}p$, $\text{EG}p$ and $\text{AG}p$ for a particular state on four simple transition systems. The computation trees to the right of the transition systems are generated by unwinding the transition relation. Note that all the paths originating from state s_3 are included in these computation trees. If we assume that s_3 is an initial state, then these paths correspond to execution paths. One nice property of the CTL is that for finite-state transition systems, finding the truth value of a CTL formula has linear time complexity in the size of the transition system and the formula.

Consider the two properties of the bakery algorithm discussed above. The safety property – two processes are never at the critical section at the same time – can be expressed in CTL as follows:

$$(B1) \quad \text{AG}(\neg(pc_1 = C \wedge pc_2 = C))$$

Now recall the progress property: If a process starts waiting for entry to the critical section, it eventually gets in. For the first process, this can be expressed as:

$$(B2) \quad \text{AG}(pc_1 = W \rightarrow \text{AF}(pc_1 = C))$$

3.1.3 CTL*

CTL* has two types of formulas: A *path formula* is a formula which is true or false for paths (all LTL formulas are path formulas). A *state formula* is a formula which is true or false for states (all CTL formulas are state formulas). I.e., truth set of a state

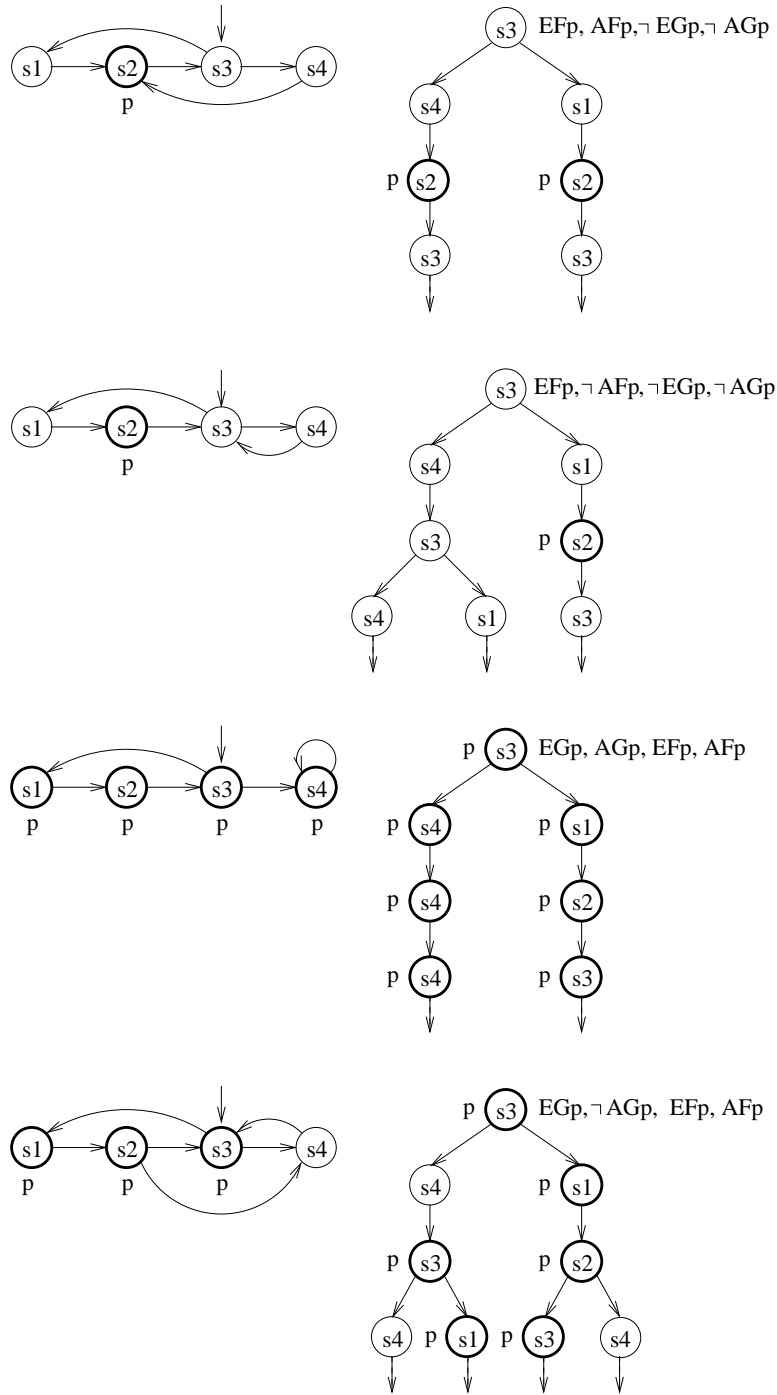


Figure 3.1: CTL Operators EF, AF, EG and AG

formula is a set of states, whereas truth set of a path formula is a set of paths. Note that all the atomic properties, i.e., formulas in AP are state formulas. We will denote the set of state formulas as SF , and the set of path formulas as PF . We will also assume that every state formula is a path formula with the interpretation that the truth of the formula depends only on the first state of the path.

Any formula in the form Gp , Fp , pUq , or Xp will be a path formula, i.e., semantics of these temporal operators are defined on paths (as in LTL). To generate state formulas from path formulas we will use the path quantifiers A and E . Given a path formula p , Ap and Ep will be state formulas with the following interpretation: Ap is true on a state, if and only if, for all paths originating from that state p is true. Ep is true on a state, if only if, there exists a path originating from that state where p is true.

As an example consider the formula Fp , where p is an atomic property. Fp will be true for a path if there exists a state on the path which satisfies p . AFp will be true for a state if for all paths originating from that state Fp is true. Similarly, EFp will be true for a state if there exists a path originating from that state for which Fp is true. After these intuitive definitions now we can formally define the syntax and semantics of CTL*.

The following grammar defines the syntax of CTL*, where terminals \neg , \wedge , and \vee are logical connectives, terminals X and U are temporal operators, terminals E and A are path quantifiers, terminal ap denotes a atomic property, (i.e., $ap \in AP$), nonterminal sf denotes a state formula, (i.e., a member of SF) and nonterminal pf denotes a path formula (i.e., a member of PF):

$$\begin{aligned} sf & ::= ap \mid sf \wedge sf \mid sf \vee sf \mid \neg sf \mid E pf \mid A pf \\ pf & ::= sf \mid pf \wedge pf \mid pf \vee pf \mid \neg pf \mid X pf \mid pf U pf \end{aligned}$$

We have the following equivalences for CTL* operators, $Fp \equiv \mathbf{true}Up$, $Gp \equiv \neg F\neg p$, and $Ap \equiv \neg E\neg p$. Based on these equivalences the set $\{\neg, \vee, E, X, U\} \cup AP$ forms a basis for CTL*.

In Table 3.3 we define the semantics of CTL*. We will also define the truth value of a CTL* formulas for transition systems. For this definition we will only use state formulas. To assert the truth value of a path formula for a transition system, we will convert it to a state formula by placing the for all path quantifier A in front of it. This is equivalent to stating that the path formula is true for all execution paths of the transition system (this is exactly the same way we defined the truth value of LTL formulas for transition systems). Definition of the truth value of CTL* formulas for transition systems is as follows:

Given a transition system $T = (S, I, R)$, and a temporal formula p , $T \models p$ if and only if for all states $s \in I$, $s \models p$.

$s \models p$	iff	$L(s, f) = \mathbf{true}$ where $p \in AP$.
$s \models p \wedge q$	iff	$s \models p$ and $s \models q$ where $p, q \in SF$.
$s \models p \vee q$	iff	$s \models p$ or $s \models q$ where $p, q \in SF$.
$s \models \neg p$	iff	not $s \models p$ where $p \in SF$.
$s \models Ep$	iff	there exists a path x such that $x_0 = s$, and $x \models p$, where $p \in PF$.
$s \models Ap$	iff	for all paths x with $x_0 = s$, $x \models p$, where $p \in PF$.
$x \models p$	iff	$x_0 \models p$, where $p \in SF$.
$x \models p \wedge q$	iff	$x \models p$ and $x \models q$ where $p, q \in PF$.
$x \models p \vee q$	iff	$x \models p$ or $x \models q$ where $p, q \in PF$.
$x \models \neg p$	iff	not $x \models p$ where $p \in PF$.
$x \models Xp$	iff	$x^1 \models p$ where $p \in PF$.
$x \models pUq$	iff	there exists an i such that $x^i \models q$, and for all $j < i$, $x^j \models p$ where $p, q \in PF$.

Table 3.3: CTL* semantics for state (SF) and path (PF) formulas.

I.e., a transition system satisfies a CTL* formula if all its initial states satisfy the formula. Note that this is exactly the same definition we used for CTL formulas.

CTL* subsumes several temporal logics including the branching time temporal logic CTL and the linear time temporal logic LTL. CTL can be defined using CTL* syntax as follows:

Any CTL* formula in which every temporal operator F, G, X and U is immediately preceded with one of the path quantifiers E or A is a CTL formula.

Similarly LTL can be defined using CTL* syntax:

Any CTL* formula in which path quantifiers E and A are never used is an LTL formula.

The semantics of CTL and LTL can also be derived from the CTL* semantics given in Table 3.3.

Although CTL and LTL form proper subsets of CTL*, a rich set of properties can be expressed in them. Still, there are some properties that can be expressed in CTL* and LTL that are not expressible in CTL. A typical example is GFp , i.e., always eventually p is true. This could be used to express *fairness*, e.g., always eventually a process is scheduled. Note that this property is not equivalent to $AGEFp$ or $AGAFp$. Actually, it cannot be expressed in CTL. Similarly, the property $AGEFp$, which could be useful for detecting deadlocks in protocols can be expressed in CTL* and CTL, but it cannot be expressed in LTL. There has been a good amount of discussion on which logic is

more appropriate for specifying properties of reactive systems. A lot of interesting properties can be expressed in both of these logics. Most symbolic model checkers (such as SMV [McM93]) use CTL, whereas most automata-based model checkers (such as SPIN [Hol97]) use LTL.

3.1.4 Fixpoint characterization of temporal formulas

All temporal properties in CTL can be expressed using only the logical connectives \neg , \vee , the existential next state operator EX, and a least fixpoint operator. The logic which includes these operators is called the μ -calculus [Koz83], and it subsumes most temporal logics, including CTL, LTL and more powerful logics such as CTL* [Dam94, Eme90]. The power of the μ -calculus comes from its fixpoint operators.

Since μ -calculus formulas are constructed from simple operators, they look quite cumbersome. However, they are closely related to the procedures used in symbolic model checking to compute the truth sets of temporal properties. Looking at a μ -calculus formula one can see how a symbolic model checker would compute it. Hence, μ -calculus formulas are very useful in explaining the symbolic model checking procedures presented below.

The following grammar defines the syntax of the μ -calculus, where terminals \neg , \wedge , and \vee are logical connectives; terminal EX is the existential next state temporal operator; terminal AX is the universal next state temporal operator; terminal μ is the least fixpoint operator; terminal ν is the greatest fixpoint operator; terminal x is a propositional variable; terminal ap denotes an atomic property, (i.e., $\text{ap} \in AP$); and nonterminal f denotes a μ -calculus formula:

$$f ::= \text{ap} \mid f \wedge f \mid f \vee f \mid \neg f \mid \text{EX } f \mid \text{AX } f \mid \mu x . f \mid \nu x . f$$

where, in the last two production rules, f must be syntactically monotone in the variable x , i.e., all free occurrences of x in f must occur in the scope of an even number of negations \neg . The notions of scope, bound and free occurrences are the same as the first order predicate logic, where fixpoint operators μ and ν are treated as quantifiers. We have the following equivalences

$$\text{AX } p \equiv \neg \text{EX } \neg p \quad \text{and} \quad \nu x . \mathcal{F} x \equiv \neg \mu x . \neg \mathcal{F} \neg x$$

Hence, we can represent any μ -calculus formula using the basis $\{\neg, \vee, \text{EX}, \mu\} \cup AP$.

To explain the interpretation of μ -calculus formulas, we need a short discussion on the lattice theory. Given a transition system $T = (S, I, R)$, the power set of S , 2^S may be viewed as a complete lattice $(2^S, S, \emptyset, \subseteq, \cup, \cap)$, with the top element S , the bottom element \emptyset , intersection \cap as the meet operator, union \cup as the join operator, and the set containment \subseteq as the partial ordering.

We can interpret every temporal logic formula as a representation of its truth set. Hence, **false** corresponds to the bottom element \emptyset of the lattice, and **true** corresponds to the top element S (we use temporal formulas and their truth sets interchangeably). Then next operators EX and AX correspond to functions from 2^S to 2^S . Therefore, given a set of states denoted by a proposition p , EX p maps p to the states which can reach p after traversing a single transition, i.e.,

$$\text{EX } p \equiv \{ s \mid \exists s' . s' \in p \wedge (s, s') \in R \}.$$

One notation for representing such functions is $\lambda x . \text{EX } x$, where x denotes the input variable which is a set of states (we drop λx when it is clear from the context).

Given a function $\mathcal{F} : 2^S \rightarrow 2^S$, $\mathcal{F} p$ denotes the application of function \mathcal{F} to set $p \subseteq S$. Also, $\mu x . \mathcal{F} x$ denotes the least fixpoint of \mathcal{F} , i.e., the smallest x such that $x \equiv \mathcal{F} x$. Similarly, $\nu x . \mathcal{F} x$ denotes the greatest fixpoint of \mathcal{F} , i.e., the greatest x such that $x \equiv \mathcal{F} x$. We call a function \mathcal{F} *monotonic*, if $p \subseteq q$ implies $\mathcal{F} p \subseteq \mathcal{F} q$. We have the following property from the lattice theory [Tar55]:

Theorem 1 *Let $\mathcal{F} : 2^S \rightarrow 2^S$ be a monotonic function. Then \mathcal{F} always has a least and a greatest fixpoint, which are respectively*

$$\begin{aligned} \mu x . \mathcal{F} x &\equiv \bigcap \{ x \mid \mathcal{F} x \subseteq x \} \\ \nu x . \mathcal{F} x &\equiv \bigcup \{ x \mid \mathcal{F} x \supseteq x \} \end{aligned}$$

Hence, $\mu x . \mathcal{F} x$ is the intersection of all the sets x where $\mathcal{F} x \subseteq x$, and similarly, $\nu x . \mathcal{F} x$ is the union of all the sets x where $\mathcal{F} x \supseteq x$. This property is valid even when S (hence the lattice) is infinite.

The semantics of the μ -calculus follows from this property, and the semantics of the next state operators defined previously. Note that the syntactic restriction on the μ -calculus formulas (all free occurrences of a variable occurs in the scope of an even number of negations) guarantees that every function appearing in a formula is monotonic, i.e., the property given above is always applicable.

Below we show the μ -calculus representation of CTL properties. The logical connectives \neg , \vee , \wedge , and the next state operators EX and AX are included in both CTL and the μ -calculus, and hence, they do not need to be translated. The CTL temporal operators EF, AF, EG, and AG can all be expressed using the CTL temporal operators EU and AU using the equivalences given in the previous section. Hence, we only need to consider CTL formulas of the form $p \text{ EU } q$, and $p \text{ AU } q$ for translation. Translation of these formulas to μ -calculus is given by the following property [Arn94]:

Theorem 2 *Truth sets of CTL formulas $p EU q$, and $p AU q$ can be expressed as least fixpoints:*

$$\begin{aligned} p EU q &\equiv \mu x . q \vee (p \wedge EX x) \\ p AU q &\equiv \mu x . q \vee (p \wedge AX x) \end{aligned}$$

Chapter 4

Symbolic Model Checking

Given a transition system $T = (S, I, R)$ and a temporal property p , we want to verify that $T \models p$. A procedure which can automatically verify this is called a *model checker*, since it is checking if the transition system T is a model for the temporal formula p . As defined above, $T \models p$ if for all initial states $s \in I$, $s \models p$. In other words, using the truth set interpretation of the temporal formulas, $T \models p$ if $I \subseteq p$.

Finding a *bug* in a system is equally valuable (if not more valuable) as verifying its properties. Finding a bug corresponds to finding an initial state $s \in I$ which does not satisfy the temporal property, i.e., $s \in I \cap \neg p$. It is possible to generate a counter-example execution path, which demonstrates violation of property p , starting from such a state. One of the reasons for success of model checking procedures is the fact that they are able to generate counter-example behaviors [CGMZ94].

Then, our goal in analyzing a system $T = (S, I, R)$ and a temporal property p is :

- Either to prove that the system T satisfies the property p by showing that $I \subseteq p$.
- Or to demonstrate a bug by finding a state $s \in I \cap \neg p$, and generating a counter-example execution path starting from s .

Assume that there exists a representation for sets of states of a transition system (i.e., subsets of S) which supports tests for equivalence, emptiness, and membership. Then, if we can represent the truth set of the temporal property p , and the set of initial states I using this representation, we can check the conditions above. If the state space is finite, explicit state enumeration would be one such representation. Note that as the state spaces of the programs grow, explicit state enumeration will become more expensive since the size of this representation is linearly related to the number of states in the set it represents. Unfortunately, state spaces of programs increase exponentially with the number of variables and concurrent components. This is called the *state space explosion problem*, and makes a naive implementation of the explicit state enumeration infeasible.

Another method is to use a *symbolic representation* for encoding sets of states. Symbolic representations are mathematical objects with semantics corresponding to sets of states. We can use such representations in encoding the truth sets of temporal formulas. For example Boolean logic formulas can be used as a symbolic representation for finite-state systems.

Model checking procedures compute a representation for truth sets of temporal formulas using state space exploration. Fixpoints corresponding to truth sets of temporal formulas can be computed by aggregating sets of states iteratively. Temporal properties which require more than one fixpoint computation can be computed recursively starting from the inner fixpoints and propagating the partial results to the outer fixpoints. In the following sections, we demonstrate how to implement a model checker, and discuss the functionality that a symbolic representation has to support to be included in a model checker.

4.0.5 Computing Truth Sets of Temporal Properties

Given a function \mathcal{F} , $\mathcal{F}^i p$ is defined as:

$$\mathcal{F}^i p \stackrel{\text{def}}{=} \underbrace{\mathcal{F}(\mathcal{F} \dots (\mathcal{F} p))}_{i \text{ times}}.$$

We define \mathcal{F}^0 as the identity relation. Then, we have the following property [Tar55]:

Theorem 3 *Given a monotonic function $\mathcal{F} : 2^S \rightarrow 2^S$, for all n ,*

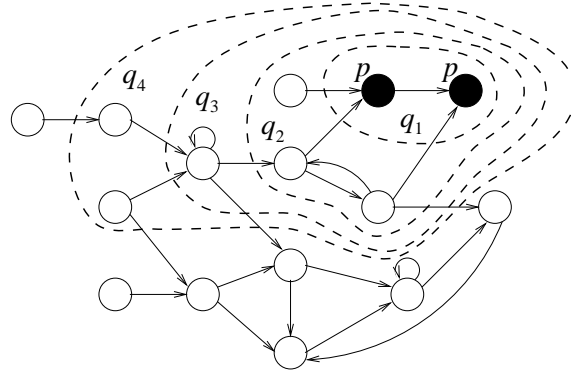
$$\begin{aligned} \mu x . \mathcal{F} x &\supseteq \bigcup_{i=0}^n \mathcal{F}^i \mathbf{false} \\ \nu x . \mathcal{F} x &\subseteq \bigcap_{i=0}^n \mathcal{F}^i \mathbf{true} \end{aligned}$$

This property holds even when the lattice is infinite.

Assume that we generate a sequence of approximations to the least fixpoint $\mu x . \mathcal{F} x$ of a monotonic function \mathcal{F} by generating the following sequence:

$$\mathbf{false}, \mathcal{F} \mathbf{false}, \mathcal{F}^2 \mathbf{false}, \dots, \mathcal{F}^i \mathbf{false}, \dots$$

This sequence is monotonically increasing since \mathbf{false} corresponds to the bottom element of the lattice, and the function \mathcal{F} is monotonic. If this sequence converges to a fixpoint, i.e., if we find an i where $\mathcal{F}^i \mathbf{false} \equiv \mathcal{F}^{i+1} \mathbf{false}$, then from the property above, we know that it is the least fixpoint, i.e., it is equal to $\mu x . \mathcal{F} x$.



$$\begin{aligned}
 \text{EF } p \equiv & \underbrace{\text{false}}_{q_0} \vee p \vee \text{EX } p \vee \text{EX } (\text{EX } p) \vee \text{EX } (\text{EX } (\text{EX } p)) \vee \dots \\
 & \underbrace{\hspace{1.5cm}}_{q_1} \\
 & \underbrace{\hspace{3.5cm}}_{q_2} \\
 & \underbrace{\hspace{6.5cm}}_{q_3} \\
 & \underbrace{\hspace{10.5cm}}_{q_4}
 \end{aligned}$$

Figure 4.1: Fixpoint Computations for EF p

Similarly, a monotonically decreasing sequence of approximations could be generated for the greatest fixpoint. In this discussion we only use least fixpoints. Because of the duality between the least and the greatest fixpoints, this does not restrict the set of properties that can be analyzed using the techniques discussed below.

As an example for computing least fixpoints, consider the property EF p . A state satisfies EF p if there exists a path starting from that state which has a state that satisfies p . Based on the properties discussed above, we have

$$\text{EF } p \equiv \mathbf{true} \quad \text{EU } p \equiv \mu x . p \vee (\mathbf{true} \wedge \text{EX } x) \equiv \mu x . p \vee \text{EX } x$$

i.e., EF p is equivalent to the least fixpoint of the function $\mathcal{F} x \equiv p \vee \text{EX } x$. We can compute the truth set of EF p by generating the following sequence:

$$\begin{aligned}
 & \underbrace{\text{false} \vee p \vee \text{EX } p}_{\mathcal{F} \text{ false}} \vee \text{EX } (\text{EX } p) \vee \text{EX } (\text{EX } (\text{EX } p)) \vee \dots \\
 & \underbrace{\hspace{3.5cm}}_{\mathcal{F}^2 \text{ false}} \\
 & \underbrace{\hspace{6.5cm}}_{\mathcal{F}^3 \text{ false}}
 \end{aligned}$$

as shown in Figure 4.1 on an example transition system. When this sequence converges to a fixpoint, the result will be equal to the truth set of the property EF p .

Procedure Evaluate(f)	
Case	
IsEvaluated(f)	: Return(f)
$f = \neg f_1$: Return(Not(Evaluate(f_1)))
$f = f_1 \wedge f_2$: Return(And(Evaluate(f_1),Evaluate(f_2)))
$f = f_1 \vee f_2$: Return(Or(Evaluate(f_1),Evaluate(f_2)))
$f = \text{EX } f_1$: Return(EX(Evaluate(f_1)))
$f = \mu x . \mathcal{F } x$: $q_0 \equiv \mathbf{false}$ $q_{i+1} \equiv \text{Evaluate}(\mathcal{F } q_i)$ Return(q_n) when Equivalent(q_n, q_{n+1})

Figure 4.2: A Model Checking Algorithm for μ -calculus

Based on these properties we can develop a μ -calculus model checker as shown in Figure 4.2. The procedure `Evaluate(f)` computes the truth set of μ -calculus formula f . Using the equivalences $\text{AX } p \equiv \neg \text{EX } \neg p$, and $\nu x . \mathcal{F } x \equiv \neg \mu x . \neg \mathcal{F } \neg x$, this procedure can compute truth set of any μ -calculus formula.

The function `IsEvaluated(f)` returns **true** if the truth set of f is already computed. Note that for a basic property $f \in BP$, `IsEvaluated(f)` always returns **true**, i.e., we assume that the truth set of basic properties are represented in the form we want before we call the procedure `Evaluate`.

As discussed in the previous section, after the truth set of f is computed, we can check if $I \subseteq f$. If this condition holds the model checker reports that the property is proved. If $I \not\subseteq f$, we compute the truth set of $\neg f$ and look for a state $s \in I \cap \neg f$, and generate a counter example. We do not discuss counter-example generation here, it is reported in [CGMZ94].

4.0.6 Symbolic Representations

Assume that `Symbolic` is the data type used for encoding sets of states, and `Formula` is the data type used for representing μ -calculus formulas. Then, the signature of the procedure `Evaluate`, shown in Figure 4.2, is

`Symbolic Evaluate(Formula)`

The procedure `Evaluate` computes a representation of type `Symbolic` for the truth set of its argument formula. Note that, in Figure 4.2, the operators \neg , \wedge , \vee , and EX are implemented by procedures `Not`, `And`, `Or`, and `EX`. We specify these procedures below:

`Symbolic Not(Symbolic)` : Given an argument that represents a set $p \subseteq S$, it returns a representation for $S - p$.

Symbolic And(Symbolic,Symbolic) : Given two arguments representing two sets $p, q \subseteq S$, it returns a representation for $p \cap q$.

Symbolic Or(Symbolic,Symbolic) : Given two arguments representing two sets $p, q \subseteq S$, it returns a representation for $p \cup q$.

Symbolic EX(Symbolic) : Given an argument that represents a set $p \subseteq S$, it returns a representation for the set $\{s \mid \exists s' . s' \in p \wedge (s, s') \in R\}$.

Boolean Equivalent(Symbolic, Symbolic) : Given two arguments representing two sets $p, q \subseteq S$, it returns **true** if $p \equiv q$, returns **false** otherwise.

Using the procedures described above, given a temporal formula, the procedure **Evaluate** computes a symbolic representation for its truth set.

The computation of the procedure **EX** involves computing a relational image. Given a set $p \subseteq S$ and a relation $X \subseteq S \times S$ we use $X p$ to denote relational image of p under X , i.e., $X p$ is defined as restricting the domain of X to set p , and returning the range of the result. Note that we can think of relation X as a functional $X : 2^S \rightarrow 2^S$. Then, $X p$ denotes the application of the functional X to set p .

Let R^{-1} denote the inverse of the transition relation R . Then $\text{EX } p \equiv R^{-1} p$, i.e., functional **EX** corresponds to the inverse of the transition relation R . Hence, we can compute the procedure **EX** using a relational image computation. Most model checkers represent transition relation R in a partitioned form to make the relational image computation more efficient [BCL91].

Any representation which can encode the set of initial states I , and the set of basic properties BP , and supports the above functionality could be used as a symbolic representation in a symbolic model checker. We call a representation L an *adequate language* [KMM⁺97] if

- Initial states I , the transition relation R , and the set of basic properties BP can be expressed in L .
- L is effectively closed under the Boolean connectives negation, conjunction and disjunction.
- Satisfiability is decidable for assertions over L , and we have an algorithm to carry out the procedure.
- There is also an algorithm to compute binary relational images over L . That is, given X of type $L \rightarrow L$, and $p \in L$, we can compute $X p$, which is defined by restricting the domain of X to set p , and returning the range of the result.

For example, for finite-state systems, Boolean logic could be used as a symbolic representation. We can implement procedures for negation, conjunction and disjunction of Boolean logic formulas. We can also implement a procedure for checking satisfiability. If we can represent the transition relation R as a Boolean logic formula, then relational image computation $R^{-1} p$ can be computed by conjuncting the formula representing R and formula representing p , and eliminating the variables in the domain of R^{-1} using existential quantifier elimination.

Ordered binary decision diagrams (BDDs) are an efficient data structure for representing Boolean logic formulas, and they support all the functionality we described above [Bry86]. They have been successfully used for symbolic model checking [BCM⁺90, McM93].

Chapter 5

Automata Theoretic Approach to Model Checking

The automata theoretic approach to model checking can be summarized as follows [VW86]: Given a temporal property f (in LTL) and a transition system T :

1. Convert the negation of the temporal property to a Büchi automaton $A_{\neg f}$ such that all the sequences which satisfy $\neg f$ should be accepted by $A_{\neg f}$.
2. Model the input transition system as a Büchi automaton A_T by labeling the transitions going out of a state with the atomic propositions that are true in that state.
3. Generate the product automata $A_T \times A_{\neg f}$ which accepts only the sequences accepted by both A_T and $A_{\neg f}$.
4. Check if the language accepted by $A_T \times A_{\neg f}$ is empty. If it is empty this proves that T satisfies the temporal property f . If it is not empty, generate an accepting sequence for $A_T \times A_{\neg f}$ which shows a counter-example behavior in T that violates f .

A Büchi automaton is an automaton $A = (\Sigma, S, \delta, S_0, F)$ where: Σ is a finite alphabet; S is a finite set of states; δ is a deterministic $\delta : S \times \Sigma \rightarrow S$ or nondeterministic $\delta : S \times \Sigma \rightarrow 2^S$ transition function; $S_0 \subseteq S$ is a set of initial states; and $F \subseteq S$ is a set of accepting states.

Büchi automata accept infinite words. An infinite word $w = a_0a_1a_2\dots$ defines a function from set of natural numbers \mathcal{N} to the alphabet Σ , i.e., $w : \mathcal{N} \rightarrow \Sigma$ where $w(i) = a_i$.

A run over automaton A over an infinite word $w = a_0a_1a_2\dots$ is a sequence of automaton states $s_0, s_1, s_2\dots$ where $s_0 \in S_0$ and for all i , $s_{i+1} \in \delta(s_i, a_i)$. A run r over w defines a function from set of natural numbers \mathcal{N} to the set of states S , i.e., $r : \mathcal{N} \rightarrow S$ where $r(0) \in S_0$ and for all i , $r(i+1) \in \delta(r(i), w(i))$.

A run r is an accepting run if there is an accepting state $s \in F$ that repeats infinitely often in r , i.e. for infinitely many i 's, $s_i = s$. In other words, $\text{inf}(r) \cap F \neq \emptyset$, where $\text{inf}(r)$ is the set of states that appear infinitely often in r .

An infinite word w is accepted by a Büchi automaton A , if there exists an accepting run of A over w . The set of infinite words accepted by A is denoted as $L(A)$ the language accepted by A .

Given an LTL property f it is possible to construct a Büchi automaton A_f such that every infinite word accepted by the automaton A_f corresponds to a path that satisfies the temporal property f [VW86, GPVW95]. In the worst-case the number of states in A_f is exponential in the length of the formula f , i.e., $|S| = O(2^{|f|})$.

Translating a transition system to a Büchi automaton is straight forward. Consider a transition system $T = (S, I, R)$ where S is the set of states, $I \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is the transition relation. The labeling function $L : S \rightarrow 2^{AP}$ maps each state $s \in S$ to a set of atomic propositions in AP . The Büchi automaton A_T that corresponds to the transition system $T = (S, I, R)$ is defined as $A_T = (2^{AP}, S, \delta, I, S)$, i.e., the alphabet of A_T is 2^{AP} , the set of states of A_T is equal to the set of states of T , the set of starting states of A_T is equal to the set of initial states of T , and the set of final states of A_T includes all the states of T . Finally, the transition function δ is defined as $s' \in \delta(s, a)$ if and only if $(s, s') \in R$ and $a = L(s)$.

If we view each path $x = s_0, s_1, s_2, \dots$ in T as a function from natural numbers to sets of atomic propositions $x : \mathcal{N} \rightarrow 2^{AP}$ where $x(i) = L(s_i)$ then the set of infinite words accepted by the Büchi automaton A_T is exactly the set of all infinite paths in T .

5.0.7 Model Checking as Language Emptiness of Büchi Automaton

Given a transition system $T = (S, I, R)$ and a LTL formula f we can construct Büchi automata A_T and $A_{\neg f}$. Note that $L(A_T)$ consists of all the paths of the transition system T , and $L(A_{\neg f})$ consists of all the paths that satisfies $\neg f$, i.e., all the paths that violate f . If we can find a path in $L(A_T) \cap L(A_{\neg f})$ then this means that there is a path in T which violates f (i.e., a counter-example behavior). However if $L(A_T) \cap L(A_{\neg f}) = \emptyset$ then this proves that all the paths in T satisfy f , i.e., $T \models f$.

To check if $L(A_T) \cap L(A_{\neg f})$ is empty we can construct the product automaton $A_T \times A_{\neg f}$ where $L(A_T \times A_{\neg f}) = L(A_T) \cap L(A_{\neg f})$. Then we can check if the language accepted by $A_T \times A_{\neg f}$ is empty.

Given $A_T = (\Sigma, S_1, \delta_1, S_{01}, F_1)$ and $A_{\neg f} = (\Sigma, S_2, \delta_2, S_{02}, F_2)$ the product automaton is defined as follows $A_T \times A_{\neg f} = (\Sigma, S, \delta, S, F)$ where $S = S_1 \times S_2$, $S_0 = S_{01} \times S_{02}$, $F = \{F_1 \times S_2, F_2 \times S_1\}$ (this generalized Büchi automaton can be converted to a Büchi automaton). The transition function δ is defined as $(s', t') \in \delta((s, t), a)$ if and only if $s' \in \delta_1(s, a)$ and $t' \in \delta_2(t, a)$.

5.0.8 Algorithm for Determining Language Emptiness of Büchi Automaton

Given a Büchi automaton $A = (\Sigma, S, \delta, S, F)$, $L(A)$ is not empty if there exists an infinite sequence s_0, s_1, s_2, \dots such that $s_0 \in S$, for all i , there exists an $a_i \in \Sigma$ such that $s_{i+1} \in \delta(s_i, a_i)$ and there exists an accepting state $s \in F$ where for infinitely many i 's, $s_i = s$. Such an infinite sequence exists if and only if there exists an accepting state $s \in F$ such that s is reachable from an initial state in S_0 and s is reachable from itself, i.e., s is contained in a cycle.

To find cycles in a graph one can use a depth-first search algorithm which constructs the strongly connected components in linear time by adding two integer numbers to every state reached. If a strongly connected component reachable from an initial state contains an accepting state then the language accepted by the Büchi automaton is not empty. There is a more memory efficient algorithm for checking the same condition which is called *nested depth first search* [CVWY92].

The main idea is as follows:

- Do a depth first search of the reachable states
- While doing this search build a postorder list (ancestors of a state should appear after it) of reachable accepting states. Let this ordered list be $Q = s_1, s_2, \dots, s_k$ where s_1 is the first postorder reachable state and s_k is the last.
- Do a second depth first search from the elements in Q
 - Start the search from s_1
 - Once the search from s_i is finished (s_i is reached, i.e., a cycle is found, or no more reachable states from s_i), restart the search from s_{i+1} but do not reconsider the states that have been visited during searches from s_j , for $j \leq i$.

This algorithm visits each state in the graph once in each depth-first search, and it only needs to mark each visited state (i.e., there is no need to store two integer variables per state, this search can be implemented using one Boolean variable per state).

We can interleave the first and the second depth first search. In the interleaved nested depth first search two Boolean variables per state are used to mark if the stored state is visited during the first search or the second search.

NESTED DEPTH FIRST SEARCH (WITHOUT INTERLEAVING)

```
main()
{
  Stack = S_0;
  Queue = {};
  StateSpace = {};
  search1();
  while Queue not empty
  {
    s = head(Queue);
    remove s from Queue;
    push s to Stack;
    seed = s;
    search2();
  }
}

search1()
{
  if Stack is empty return();
  s = top(Stack);
  add (s,1) to StateSpace;
  if error(s) report_error();
  else
  {
    for each successor t of s do
      if (t,1) not in StateSpace
      {
        push t to Stack;
        search1();
      }
  }
  if accepting(s) add s to Queue;
  delete s from Stack;
}

search2()
{
  if Stack is empty return();
```

```
s := top(Stack);
add (s,2) to StateSpace;
for each successor t of s do
  if (t,2) not in StateSpace
  {
    push t to Stack;
    search2();
  }
  else
    if (t == seed) report_cycle();
remove s from Stack;
}
```

NESTED DEPTH FIRST SEARCH (WITH INTERLEAVING)

```
main()
{
  Stack1 = S_0;
  Stack2 = {};
  StateSpace = {};
  search1();
}

search1()
{
  if Stack1 is empty return();
  s = top(Stack1);
  add (s,1) to StateSpace;
  if error(s) report_error();
  else
  {
    for each successor t of s do
      if (t,1) not in StateSpace
      {
        push t to Stack1;
        search1();
      }
  }
  if accepting(s)
  {
    seed = s;
    push s to Stack2;
    search2();
  }
  remove s from Stack1;
}

search2()
{
  if Stack2 is empty return();
  s = top(Stack2);
  add (s,2) to StateSpace;
  for each successor t of s do
```

38

```
    if (t,2) not in StateSpace
    {
        push t to Stack2;
        search2();
    }
    else
        if (t == seed) report_cycle();
remove s from Stack2;
}
```

Bibliography

- [ACH⁺95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, X. Nicollin, P. H. Ho, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
- [AHH96] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.
- [Arn94] A. Arnold. *Finite Transition Systems: Semantics of Communicating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.
- [BCL91] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Proceedings of the International Conference on Very Large Scale Integration*, August 1991.
- [BCM⁺90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. H. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, January 1990.
- [BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.
- [BGP99] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.

- [BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder - a second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification*, 2000.
- [BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of PLDI*, volume 36-5 of *SIGPLAN Notices*, pages 203–213, 2001.
- [BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstractions for model checking c programs. In *Proceedings of TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001.
- [BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN Workshop on Model Checking Software*, volume 2057 of *LNCS*, pages 103–122, 2001.
- [Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [CAB⁺98] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.
- [CBH⁺00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasarenau, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. 22nd Int. Conf. on Soft. Eng. (ICSE)*, 2000.
- [CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, June 1989.
- [CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
- [CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1981.

- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
- [CGMZ94] E. M. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. Technical Report CMU-CS-94-204, School of Computer Science, Carnegie Mellon University, October 1994.
- [CK96] E. M. Clarke and R. P. Kurshan. Computer aided verification. *IEEE Spectrum*, pages 61–67, June 1996.
- [CMB90] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *Proceedings of the 2nd International Conference on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 23–32. Springer, June 1990.
- [CVWY92] C. Courcobetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
- [Dam94] M. Dam. CTL* and ECTL* as fragments of the modal μ -calculus. *Theoretical Computer Science*, 126:77–96, 1994.
- [Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, 1976.
- [DP99] G. Delzanno and A. Podelski. Model checking in CLP. In Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 223–239. Springer, March 1999.
- [DP01] Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *Journal of Software Tools and Technology Transfer*, 3(3):250–270, 2001.
- [Eme90] E. A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

- [Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of the Symposium in Applied Mathematics 19*, pages 19–32, 1967.
- [God94] P. Godefroid. *Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem*. PhD thesis, Universite De Liege, 1994.
- [God97a] P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of the 9th Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 476–479. Springer-Verlag, June 1997.
- [God97b] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, January 1997.
- [GPVW95] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV 1995 Conference*, 1995.
- [Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.
- [GS97] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, June 1997.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2000.
- [KMM⁺97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.

- [Koz83] D. Kozen. Results on the propositional μ -calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- [McM93] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.
- [Pnu77] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science*, 1977.
- [Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
- [QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1982.
- [Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
- [VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of First IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.