in this chapter we consider only Büchi automata. Algorithms for other types of automata can be derived in a similar fashion from results in [60]. In general, checking language inclusion between two nondeterministic ω-automata is PSPACE-hard. For this reason we consider a restricted case of the general problem in which the specification automaton is deterministic. Thus, our algorithm cannot be used in those cases where the specification cannot be expressed using a deterministic automaton (see Section 9.2.1). For simplicity we also require that both automata are complete.

Let $\mathcal{A} = (\Sigma, Q, \Delta, Q^0, F)$ and $\mathcal{A}' = (\Sigma, Q', \Delta', Q^{0'}, F')$ be two Büchi automata over the same alphabet Σ. Let $M(\mathcal{A}, \mathcal{A}')$ be a Kripke structure $(Q \times Q', R, L)$ over $AP = \{q, q'\}$, where q, q' are two new symbols and

$q \in L((s, s'))$ iff $s \in F$.

$q' \in L((s, s'))$ iff $s' \in F'$.

$(s, s')R(r, r')$ iff $\exists \sigma \in \Sigma : (s, \sigma, r) \in \Delta$ and $(s', \sigma, r') \in \Delta'$.

Recall that in Section 5.2 we showed how to encode Kripke structures symbolically. In [60], it is shown that, if $\mathcal{A}'$ is deterministic,

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}') \Leftrightarrow M(\mathcal{A}, \mathcal{A}') \models \mathbf{A}(\mathbf{GF}q \Rightarrow \mathbf{GF}q')$$

Note that the formula above is not a CTL formula, in that there are temporal operators that are not immediately preceded by path quantifiers. However, it is equivalent to **AG AF** q' ("infinitely often q'") under the fairness constraint "infinitely often q." Checking the above formula with the given fairness constraint can be handled using the techniques described in Section 6.2.

THEOREM 8 $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$ if and only if $M(\mathcal{A}, \mathcal{A}') \models$ **AG AF** q' with fairness constraint q.

10 Partial Order Reduction

The *partial order reduction* is aimed at reducing the size of the state space that needs to be searched by model checking algorithms. It exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. Thus, this reduction technique is best suited for asynchronous systems (in synchronous systems, concurrent transitions are executed simultaneously rather than being interleaved).

The method consists of constructing a reduced state graph. The full state graph, which may be too big to fit in memory, is never constructed. The behaviors of the reduced graph are a subset of the behaviors of the full state graph. The justification of the reduction method shows that the behaviors that are not present do not add any information. More precisely, it is possible to define an equivalence relation among behaviors such that the checked property cannot distinguish between equivalent behaviors. If a behavior is not present in the reduced state graph, then an equivalent behavior must be included.

The name *partial order reduction* has its justification in early versions of the algorithms that were based on the partial order model of program execution [126, 153, 244]. However, the method can be described better as *model checking using representatives* [210, 212], since the verification is performed using representatives from the equivalence classes of behaviors.

In this chapter the *transitions* of a system play a significant role. The partial order reduction is based on the *dependency relation* that exists between the transitions of a system. Furthermore, this reduction method specifies which transitions should be included in the reduced model and which should not. As in Chapter 7, we want to distinguish between different transitions in a system. Thus, we modify the definition of a Kripke structure slightly. Instead of having one transition relation R, we will now have a *set* of transition relations T. For simplicity, we will refer to each element α in T as a *transition*, instead of a transition relation.

A *state transition system* is a quadruple (S, T, S_0, L) where the set of states S, the set of initial states S_0, and the labeling function L are defined as for Kripke structures, and T is a set of transitions such that for each $\alpha \in T$, $\alpha \subseteq S \times S$. A Kripke structure $M = (S, R, S_0, L)$ may be obtained by defining R so that $R(s, s')$ holds when there exists a transition $\alpha \in T$ such that $\alpha(s, s')$.

For a transition $\alpha \in T$, we say that α is *enabled* in a state s if there is a state s' such that $\alpha(s, s')$ holds. Otherwise, α is *disabled* in s. The set of transitions enabled in s is $enabled(s)$. A transition α is *deterministic* if for every state s there is at most one state s' such that $\alpha(s, s')$. When α is deterministic we often write $s' = \alpha(s)$ instead of $\alpha(s, s')$. Henceforth, we will only consider deterministic transitions.

A *path* from a state s in a state transition system is a finite or infinite sequence defined as follows. $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ such that $s = s_0$ and for every i, $\alpha_i(s_i, s_{i+1})$ holds. Here, we do not require paths to be infinite. Moreover, any prefix of a path is also a path. If

π is finite, then the *length* of π is the number of transitions in π and will be denoted by $|\pi|$.

10.1 Concurrency in Asynchronous Systems

A common observation about concurrent asynchronous systems is that the interleaving model imposes an arbitrary ordering between concurrent events. To avoid discriminating against any particular ordering, the events are interleaved in all possible ways. The ordering between independent transitions is largely meaningless. However, common specification languages, including many temporal logics, can distinguish between behaviors that only differ in this manner. Our aim is to take advantage of the cases where the specifications do not distinguish between such behaviors. In these cases, the partial order reduction only checks a subset of the behaviors. However, it checks sufficiently many of them to guarantee the soundness of the verification.

Putting concurrent events in various possible orderings is a potential cause of the state explosion problem. To see this, consider n transitions that can be executed concurrently. In this case, there are $n!$ different orderings and 2^n different states (one state for each subset of the transitions). If the specification does not distinguish between these sequences, it is clearly beneficial to consider only one sequence, with $n + 1$ states. This is demonstrated in Figure 10.1 with $n = 3$.

Our aim is to reduce the number of states that are considered in the model checking process, while preserving the correctness of the checked property. We will assume for

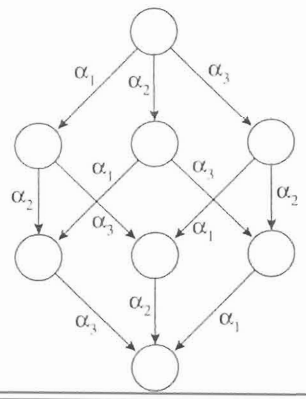

Figure 10.1
Executing three independent transitions.

simplicity of presentation that a *reduced state graph* is first generated explicitly using DFS. The model checking algorithm is then applied to the resulting state graph. The reduction constructs a graph with fewer states and edges. This speeds up the construction of the graph and uses less memory, thus resulting in a more efficient model checking algorithm. Moreover, the reduction can be applied *on-the-fly* while doing the model checking [209]. The DFS can also be replaced by breadth first search [55] and combined with symbolic model checking [4, 164].

The reduction is performed by modifying the DFS used to construct the state graph, as in Figure 10.2. The search starts with an initial state s_0 (line 1) and proceeds recursively. For each state s it selects only a subset $ample(s)$ of the enabled transitions $enabled(s)$ (in line 5), rather than the full set of enabled transitions, as in the full state space construction. The DFS explores only successors generated by these transitions (lines 6–16). In the DFS algorithm in Figure 10.2, a state is labeled as *on_stack* (lines 2,12) when it is first encountered and as *completed* (line 17) when all of its successors have been searched. Thus, a state is marked *on_stack* when it is on the DFS search stack. This information is useful for computing the function *ample*.

```
1   hash(s0);
2   set on_stack(s0);
3   expand_state(s0);

4   procedure expand_state(s)
5       work_set(s) := ample(s);
6       while work_set(s) is not empty do
7           let α ∈ work_set(s);
8           work_set(s) := work_set(s) \ {α};
9           s' := α(s);
10          if new(s') then
11              hash(s');
12              set on_stack(s');
13              expand_state(s');
14          end if;
15          create_edge(s, α, s');
16      end while;
17      set completed(s);
18  end procedure
```

Figure 10.2
Depth-first search with partial order reduction.

When the model checking algorithm is applied to the reduced state graph it terminates with a positive answer when the property holds for the original full state graph. Otherwise, it produces a counterexample. Because the reduced state graph contains fewer behaviors, the counterexample can differ from the one that would have resulted from using the full state graph.

Notice that the algorithm in Figure 10.2 constructs the reduced state graph *directly*. Constructing the full state graph and later reducing it would defy the purpose of the reduction.

In order to implement the algorithm we must find a systematic way of calculating $ample(s)$ for any given state s. The calculation of $ample(s)$ needs to satisfy three goals:

1. When $ample(s)$ is used instead of $enabled(s)$, sufficiently many behaviors must be present in the reduced state graph so that the model checking algorithm gives correct results.
2. Using $ample(s)$ instead of $enabled(s)$ should result in a significantly smaller state graph.
3. The overhead in calculating $ample(s)$ must be reasonably small.

10.2 Independence and Invisibility

In this section, we will define two concepts that can assist in reducing the state graph. As noted earlier, in the interleaving model for concurrent systems, transitions that can be executed concurrently from some state are interleaved in either order. This can be formulated by defining an independence relation on pairs of transitions that can execute concurrently. An *independence* relation $I \subseteq T \times T$ is a symmetric, antireflexive relation, satisfying the following two conditions for each state $s \in S$ and for each $(\alpha, \beta) \in I$:

Enabledness If $\alpha, \beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$.

Commutativity $\alpha, \beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

The *dependency* relation D is the complement of I, namely

$D = (T \times T) \setminus I$.

The enabledness condition states that a pair of independent transitions do not *disable* one another. Note, however, that it is possible for one to *enable* another. Note that the definition makes use of the fact that I is symmetric. The commutativity condition, which is well defined due to the enabledness condition, states that executing independent transitions in either order results in the same state. These conditions are illustrated in Figure 10.3.

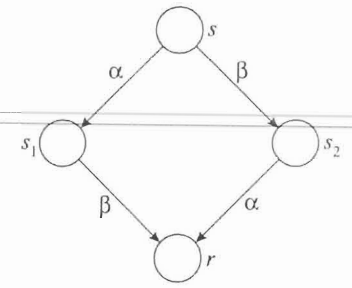

Figure 10.3
Execution of independent transitions.

When it is hard to check whether two transitions α and β are independent or not, assuming that they are dependent always preserves the correctness of the reductions described in this chapter.

The definition of independence can be used for the reduction even when two independent transitions cannot actually be executed in parallel. For example, when two transitions of different processes increment a shared variable, they satisfy the independence conditions, although some type of physical arbitration must be used to prevent them from executing simultaneously.

The commutativity condition, illustrated in Figure 10.3, suggests a potential reduction to the state graph, for it does not matter whether α is executed before β or vice versa in order to reach the state r from s. Thus, it is tempting to select only one of the transitions originating from s. This is not appropriate for the following reasons:

PROBLEM 1 The checked property might be sensitive to the choice between the states s_1 and s_2, not only the states s and r.

PROBLEM 2 The states s_1 and s_2 may have other successors in addition to r, which may not be explored if either is eliminated.

We will return to these problems at the end of Section 10.3. The first step in solving them is to define what it means for a transition to be invisible.

Let $L : S \to 2^{AP}$ be the function that labels each state with a set of atomic propositions. A transition $\alpha \in T$ is *invisible* with respect to a set of propositions $AP' \subseteq AP$ if for each pair of states $s, s' \in S$ such that $s' = \alpha(s)$, $L(s) \cap AP' = L(s') \cap AP'$. In other words, a transition is invisible when its execution from any state does not change the value of the propositional variables in AP'. A transition is *visible* if it is not invisible.

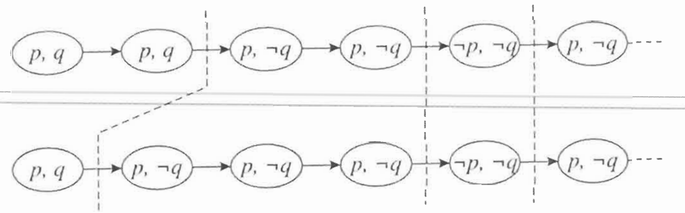

Figure 10.4
Two stuttering equivalent paths.

A closely related concept is that of *stuttering* [167], which refers to a sequence of identically labeled states along a path in a Kripke structure. Two infinite paths $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ and $\rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$ are *stuttering equivalent* (see Figure 10.4), denoted $\sigma \sim_{st} \rho$ if there are two infinite sequences of positive integers $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$ such that for every $k \geq 0$,

$$L(s_{i_k}) = L(s_{i_k+1}) = \cdots = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = \cdots = L(r_{j_{k+1}-1}).$$

We call a finite sequence of identically labeled states a *block*. Intuitively, two paths are stuttering equivalent when they can be partitioned into infinitely many blocks, such that the states in the kth block of one are labeled the same as the states in the kth block of the other. Note that corresponding blocks may have different lengths. Stuttering equivalence can be defined in a similar way for finite paths using finite sequences of indexes $0 = i_0 < i_1 < i_2 < \ldots i_n$ and $0 = j_0 < j_1 < j_2 < \ldots j_n$. Stuttering is a particularly important concept for asynchronous systems because there is no correlation between the time separating two events and the number of transitions occurring between them.

An LTL formula $\mathbf{A}f$ is *invariant under stuttering* if and only if for each pair of paths π and π' such that $\pi \sim_{st} \pi'$,

$$\pi \models f \quad \text{if and only if} \quad \pi' \models f.$$

We denote the subset of the logic LTL without the next time operator by LTL_{-X}.

THEOREM 9 Any LTL_{-X} property is invariant under stuttering.

The theorem is proved using a simple induction on the size of the LTL formula. It is interesting to note that the converse of Theorem 9 also holds [211]:

THEOREM 10 Every LTL property that is stuttering closed can be expressed in LTL_{-X}.

We now extend the notion of stuttering equivalence to structures. Two structures M and M' are *stuttering equivalent* if and only if

- M and M' have the same set of initial states.
- For each path σ of M that starts from an initial state s of M there exists a path σ' of M' from the same initial state s such that $\sigma \sim_{st} \sigma'$, and
- for each path σ' of M' that starts from an initial state s of M' there exists a path σ of M from the same initial state s such that $\sigma' \sim_{st} \sigma$.

The following corollary is useful for showing that an LTL_{-X} formula does not distinguish between structures that are stuttering equivalent. It will be exploited later, for the partial order reduction generates a structure that is stuttering equivalent to the full state graph.

COROLLARY 2 Let M and M' be two stuttering equivalent structures. Then, for every LTL_{-X} property $\mathbf{A}f$, and every initial state $s \in S_0$, $M, s \models \mathbf{A}f$ if and only if $M', s \models \mathbf{A}f$.

Returning to Figure 10.3, suppose that at least one transition, say α, is invisible, then $L(s) = L(s_1)$ and $L(s_2) = L(r)$. Consequently,

$$s\, s_1\, r \sim_{st} s\, s_2\, r$$

10.3 Partial Order Reduction for LTL_{-X}

When the specification is invariant under stuttering, commutativity and invisibility allow us to avoid generating some of the states. Based on this observation, we suggest a systematic way of selecting an ample set of transitions for any given state. The ample sets will be used by the DFS algorithm to construct a reduced state graph so that for every path not considered by the DFS algorithm there is a stuttering equivalent path that is considered. This guarantees that the reduced state graph is stuttering equivalent to the full state graph.

We say that state s is *fully expanded* when $ample(s) = enabled(s)$. In this case, all of the successors of that state will be explored by the DFS algorithm.

Instead of giving a specific algorithm for constructing ample sets, we will first provide four conditions for selecting $ample(s) \subseteq enabled(s)$ such that the satisfaction of the LTL_{-X} specification is preserved. The reduction will depend on the set of propositions AP' that appear in the LTL_{-X} formula.

Condition **C0** guarantees that if the state has at least one successor, then the reduced state graph also contains a successor for this state.

C0 $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$.

Condition **C1** is the most complicated among the constraints on $ample(s)$.

C1 [126, 153, 208, 244] Along every path in the full state graph that starts at s, the following condition holds: a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occurring first.

Note that Condition **C1** refers to paths in the *full* state graph. We need a way of checking that **C1** holds without actually constructing the full state graph. Later, we will show how to restrict **C1** so that $ample(s)$ can be calculated based on the current state s.

LEMMA 24 The transitions in $enabled(s) \setminus ample(s)$ are all independent of those in $ample(s)$.

Proof Let $\gamma \in enabled(s) \setminus ample(s)$. Suppose that $(\gamma, \delta) \in D$, where $\delta \in ample(s)$. Because γ is enabled in s, in the full graph there is a path starting with γ. But then a transition dependent on some transition in $ample(s)$ is executed before a transition in $ample(s)$, contradicting Condition **C1**. □

In order to guarantee the correctness of the DFS reduction algorithm, we need to know that if we always choose the next transition to explore from $ample(s)$, we do not omit any paths that are essential for checking the correctness of the state graph. Condition **C1** implies that such a path will have one of two forms:

- The path has a prefix $\beta_0\beta_1 \ldots \beta_m\alpha$, where $\alpha \in ample(s)$ and each β_i is independent of all transitions in $ample(s)$ including α.
- The path is an infinite sequence of transitions $\beta_0\beta_1 \ldots$ where each β_i is independent of all transitions in $ample(s)$.

Condition **C1** also implies that, if along a finite sequence of transitions $\beta_0\beta_1 \ldots \beta_m$ executed from s, none of the transitions in $ample(s)$ have occurred, then all the transitions in $ample(s)$ remain enabled. This is because each β_i is independent of the transitions in $ample(s)$ and, therefore, cannot disable them.

In the first case, assume that the sequence of transitions $\beta_0\beta_1 \ldots \beta_m\alpha$ reaches a state r. This sequence will not be considered by the DFS algorithm. However, by applying the enabledness and commutativity conditions m times, we can construct a finite sequence $\alpha\beta_0\beta_1 \ldots \beta_m$, that also reaches r. This is illustrated in Figure 10.5. In other words, even if the reduced state graph does not contain the sequence $\beta_0\beta_1 \ldots \beta_m\alpha$ that reaches the state r, we can still construct from s another sequence that reaches the same state r.

Consider the two sequences of states $\sigma = s_0s_1 \ldots s_mr$ and $\rho = sr_0r_1 \ldots r_m$ in Figure 10.5, generated by $\beta_0\beta_1 \ldots \beta_m\alpha$ and $\alpha\beta_0\beta_1 \ldots \beta_m$, respectively. In order to discard σ, we want σ and ρ to be stuttering equivalent. This is guaranteed if α is invisible, for then

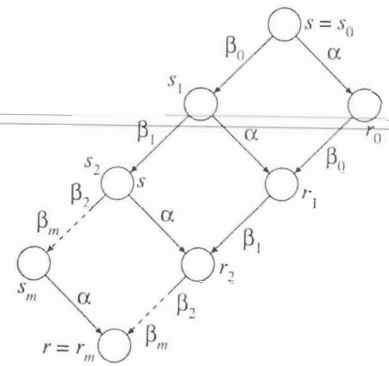

Figure 10.5
Transition α commutes with $\beta_0\beta_1 \ldots \beta_m$.

$L(s_i) = L(r_i)$ for $0 \le i \le m$. Thus, the checked property will not be able to distinguish between the two sequences above. This can be achieved by condition **C2**:

C2 [Invisibility [209]] If s is not fully expanded, then every $\alpha \in ample(s)$ is invisible.

Consider now the second case, in which an infinite path $\beta_0\beta_1\beta_2 \ldots$ that starts at s does not include any transition from $ample(s)$. By Condition **C2** all transitions in $ample(s)$ are invisible. Let α be such a transition in $ample(s)$, then the path generated by the infinite sequence of transitions $\alpha\beta_0\beta_1\beta_2 \ldots$ is stuttering equivalent to the one generated by $\beta_0\beta_1\beta_2 \ldots$. Again, even though the path $\beta_0\beta_1\beta_2 \ldots$ is not included in the reduced state graph, there is a stuttering equivalent path that is included.

Conditions **C1** and **C2** are not yet sufficient to guarantee that the reduced state graph is stuttering equivalent to the full state graph. In fact, there is a possibility that some transition will actually be delayed forever because of a cycle in the constructed state graph. As an example, consider the processes in Figure 10.6. Assume that the transition β is independent of the transitions α_1, α_2, and α_3. The transitions α_1, α_2, and α_3 are interdependent. The process on the left can execute the visible transition β exactly once. Assume there is one proposition p, which is changed from *True* to *False* by β, so that β is visible. The process on the right performs the invisible transitions α_1, α_2, and α_3 repeatedly in a loop.

The full state graph of the system in Figure 10.6 is shown on the left in Figure 10.7. The right side of the figure shows the first stages of constructing the reduced state graph, where α_1, α_2, and α_3 are invisible. Starting with the initial state s_1, we can select $ample(s_1) = \{\alpha_1\}$. Conditions **C0**, **C1**, and **C2** are satisfied. Thus, we generate $s_2 = \alpha_1(s_1)$. Similarly, we can select $ample(s_2) = \{\alpha_2\}$, generating $s_3 = \alpha_2(s_2)$. Finally, reaching s_3, Conditions **C0**, **C1**,

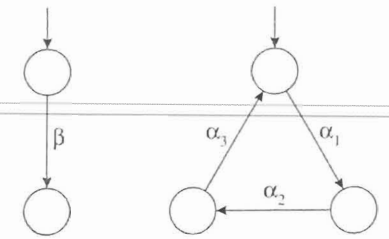

Figure 10.6
Two concurrent processes.

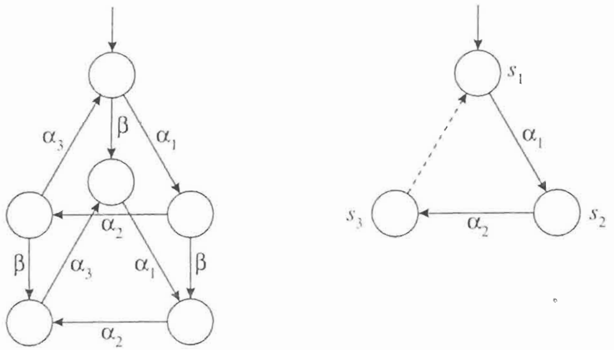

Figure 10.7
Full and reduced state graph.

and **C2** allow selecting $ample(s_3) = \{\alpha_3\}$. But the reduced state graph generated in this way does not contain any sequences where p is changed from *True* to *False*. The problem is that each state along the cycle s_1, s_2, s_3, s_1 has deferred β to a possible future state. When the cycle is closed, the construction terminates, and transition β is ignored.

To remedy this problem, we add the following condition:

C3 [Cycle condition [21, 55, 208]] A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in $ample(s)$ for any state s on the cycle.

We are now able to address Problems 1 and 2 described in the previous section. Consider Figure 10.3 again. Assume that the DFS reduction algorithm chooses β as $ample(s)$ and does not include state s_1 in the reduced graph.

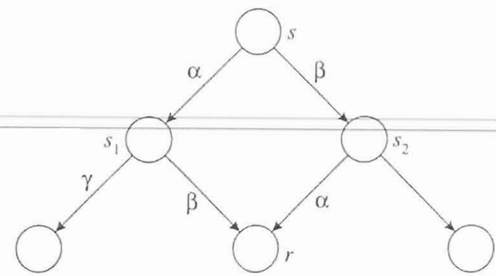

Figure 10.8
Diagram illustrating Problem 2.

We consider Problem 1 first. By Condition **C2**, β must be invisible, thus s, s_2, r and s, s_1, r are stuttering equivalent. In this chapter we are only interested in properties that are invariant under stuttering. Such properties will not be able to distinguish between the two sequences.

We next consider Problem 2. Assume that there is a transition γ enabled from s_1, as in Figure 10.8. We show that γ is still enabled at state r. Moreover, the transition sequences α, γ and β, α, γ lead to stuttering equivalent state sequences. We first note that γ cannot be dependent on β. Otherwise, the sequence α, γ violates Condition **C1**, since a transition dependent on β is executed before β. Thus, γ is independent of β. Because it is enabled in s_1, it must also be enabled in state r. Assume that γ, when executed from r, results in state r' and when executed from s_1 results in state s_1'. Since β is invisible, the two state sequences s, s_1, s_1' and s, s_2, r, r' are stuttering equivalent. Therefore, properties that are invariant under stuttering will not distinguish between the two.

10.4 An Example

Consider the mutual exclusion program P, presented in Chapter 2. The state graph for P is given in Figure 10.9. The states of the program are labeled with $AP = \{NC_i, CR_i, l_i, turn = i, \perp \mid i = 0, 1\}$, where $CR_i \in L(s)$ if $pc_i = CR_i$ in the state s, and $CR_i \notin L(s)$ if $pc_i \neq CR_i$ in s. The labeling $L(s)$ is defined similarly for all other atomic propositions in AP.

Let $f = \mathbf{G}\,\neg(CR_0 \wedge CR_1)$ be an LTL$_{-X}$ formula describing the mutual exclusion property. We will show how the DFS algorithm of Figure 10.2 can be used to construct a reduced state graph that is stuttering equivalent to the full state graph with respect to a

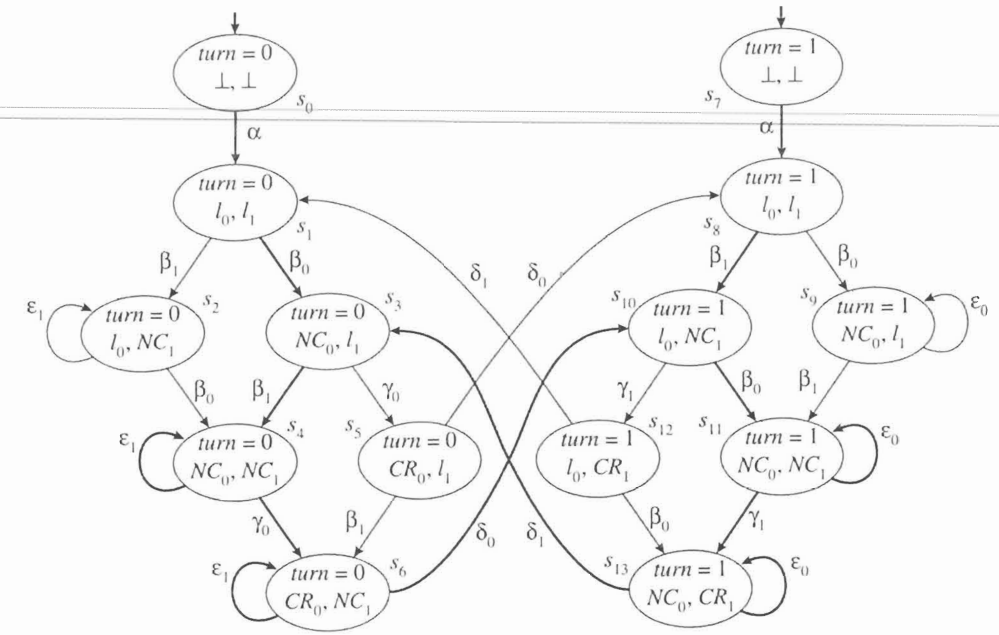

Figure 10.9
Reduced state graph for a mutual exclusion program.

subset AP' of the atomic propositions. Because we are interested in checking whether P satisfies f, we choose $AP' = \{CR_0, CR_1\}$.

Following is a list of the transitions of the program P that are enabled in some reachable state of P, where $i = 0, 1$. For brevity we omitted $same(pc_j)$ for $j \neq i$ from each of the transitions.

α: $pc = m \wedge pc_0' = l_0 \wedge pc_1' = l_1 \wedge pc' = \bot$

β_i: $pc_i = l_i \wedge pc_i' = NC_i \wedge True \wedge same(turn)$

γ_i: $pc_i = NC_i \wedge pc_i' = CR_i \wedge turn = i \wedge same(turn)$

δ_i: $pc_i = CR_i \wedge pc_i' = l_i \wedge turn' = (i+1) \bmod 2$

ε_i: $pc_i = NC_i \wedge pc_i' = NC_i \wedge turn \neq i \wedge same(turn)$

The visible transitions with respect to AP' are those in which CR_0 or CR_1 has different values before and after the transition. Thus, $\{\gamma_0, \gamma_1, \delta_0, \delta_1\}$ are visible.

Each transition is dependent on itself because the dependency relation is reflexive. All of the transitions are dependent on α since it must be executed before any other transition in the program. The dependency relation for the remaining transitions is calculated using the following two rules:

- Two transitions that change the same variable (including the program counters) are dependent.
- If one transition sets a variable and the other checks that variable, then the transitions are dependent.

Thus, all of the transitions in the same process are interdependent. Also, (γ_1, δ_0), (γ_0, δ_1), $(\varepsilon_1, \delta_0)$, $(\varepsilon_0, \delta_1)$, (δ_0, δ_1) are in D since δ_i changes the variable $turn$, while γ_i and ε_i check its value. Finally, we complete the relation D to be symmetric.

Figure 10.9 shows the full state graph. The states and edges included in the reduced state graph are shown using thick lines. Following are the states of the reduced state graph in the order they are visited by the DFS algorithm: $s_0, s_1, s_3, s_4, s_6, s_{10}, s_{11}, s_{13}, s_7, s_8$.

The DFS algorithm starts with s_0, which is one of the two initial states. For this state, $ample(s_0) = enabled(s_0) = \{\alpha\}$. For s_1, it is possible to select as $ample(s_1)$ either $\{\beta_0\}$, $\{\beta_1\}$ or $\{\beta_0, \beta_1\}$. The latter will usually result in a smaller reduction and therefore will not be considered. The first choice corresponds to selecting the enabled transitions of P_0, whereas the second choice corresponds to selecting P_1. Condition **C0** is trivially satisfied. In both cases, **C1** is satisfied. For example, suppose $ample(s_1) = \{\beta_0\}$ then along all paths leaving s_1, either β_0 is immediately executed or β_1 is executed before β_0. However, β_1 is independent of β_0.

Condition **C2** is also satisfied, for β_0 and β_1 are invisible. Finally, **C3** is satisfied because no cycle is yet formed. The choice between the two sets is arbitrary, although one may provide a better reduction in a later stages of the algorithm. We select $ample(s_1) = \{\beta_0\}$.

Executing β_0 from s_1 results in the state s_3. By using a similar argument, we select as $ample(s_3)$, the transitions of P_1 that are enabled in s_3, namely $\{\beta_1\}$. Next, we select $ample(s_4) = \{\gamma_0, \varepsilon_1\}$. We cannot select for s_4 the set $\{\gamma_0\}$, since γ_0 is visible. We cannot also select the singleton $\{\varepsilon_1\}$, because this will construct a self loop on which the transition γ_0 is enabled but never included in an ample set, thus violating Condition **C3**.

We can now select, $ample(s_6) = \{\varepsilon_1, \delta_0\}$. Because they are dependent we have to choose both in order not to violate Condition **C1**. For states s_{10} and s_{11} we choose $ample(s_{10}) = \{\beta_0\}$ and $ample(s_{11}) = \{\gamma_1, \varepsilon_0\}$. The arguments are similar to the ones for states s_3 and s_4, respectively. We next select $ample(s_{13}) = \{\delta_1, \varepsilon_0\}$. The transition δ_1 taken from s_{13} closes the cycle $s_3\, s_4\, s_6\, s_{10}\, s_{11}\, s_{13}$. By examining Figure 10.9 it is easy to check that Condition **C3** is satisfied for this cycle.

The DFS algorithm continues the search from the other initial state s_7. We select $ample(s_7) = \{\alpha\}$. Based on arguments similar to those for s_1, we also select $ample(s_8) = \{\beta_1\}$. By executing β_1 from s_8, we reach only the state s_{10} that has already been visited. Thus, the algorithm terminates.

A model-checking algorithm for LTL can now be applied to check if the reduced state graph constructed by the algorithm satisfies the formula f because $f \in \mathrm{LTL}_{-X}$. The full state graph satisfies the formula if and only if the reduced state graph does.

10.5 Calculating Ample Sets

10.5.1 The Complexity of Checking the Conditions

In order to make the partial order reduction efficient, we need to be able to calculate the ample sets for the states in the reduced graph with minimal overhead. We will consider the related problem of checking Conditions **C0** to **C3** for a set of enabled transitions at a given state. Condition **C0** for a particular state can be checked in constant time. Condition **C2** is also simple to check, by examining the transitions in the set.

Condition **C1** is a constraint that is not immediately checkable by examining the current state of the search, in that it refers to future states (some of which need not even be in the reduced state graph). The next theorem shows that, in general, checking **C1** is at least as hard as searching the full state space.

THEOREM 11 Checking Condition **C1** for a state s and a set of transitions $T \subseteq enabled(s)$ is at least as hard as checking REACHABILITY for the full state space.

Proof Consider checking whether a state r is reachable in a transition system $\mathcal{T}$ from an initial state s_0. We will reduce this problem to deciding condition **C1**. First, let α and β be new transitions. Let the transition α be only enabled at the state r. Let the transition β be enabled from the initial state and independent of all the transitions of $\mathcal{T}$. We construct β and α so that they are dependent (e.g., they both change the value of the same variable).

Consider $\{\beta\}$ as a candidate for being an ample set from s_0. First assume that **C1** is violated. Then there is a path in the new state graph along which α is performed before β. Because α is enabled only in r, this path leads from s_0 to r. The sequence of transitions on the path from s_0 to r exists also in the original state graph, in that it does not include the added transitions α or β. Thus r is reachable from s_0 in the original system.

For the other direction, assume that r is reachable in the original state graph from s_0. Then, there is a sequence from s_0 to r, which does not include β. This sequence also appears in the new state graph, and now can be extended by the transition α taken from r. The resulting sequence violates **C1**. □

In view of the previous theorem, we will avoid checking Condition **C1** for an arbitrary subset of enabled transitions. In Section 10.5.2 we will give a procedure to compute a set of transitions that is guaranteed by construction to satisfy **C1**. Although the procedure may not lead to ample sets that achieve the greatest possible reduction, it is quite efficient. There is evidently a tradeoff between efficiency of computation and the amount of reduction.

Condition **C3** is also defined in global terms. However, it refers to the reduced state graph, whereas **C1** refers to the full state graph. A possible way of implementing this constraint is to first generate a reduced state graph and then to *correct* it by adding additional transitions until it satisfies **C3** [244]. On the other hand, the approach we take replaces **C3** by a stronger condition that can be checked directly on the current state.

LEMMA 25 A sufficient condition for **C3** is that at least one state along each cycle is fully expanded.

Proof Assume there is a cycle with a fully expanded state, but the cycle does not satisfy Condition **C3**. Thus, we have some transition α that is enabled in some state s of the cycle but is never included in an ample set along the cycle. By lemma 24, if α is not included in an ample set then it is independent of all the transitions in it. Thus, α is independent of all transitions in the ample sets selected along the cycle. Consequently, it remains enabled in all the states along the cycle. However, if one of the states s' is fully expanded, meaning that $ample(s') = enabled(s')$, α is necessarily included in $ample(s')$. This contradicts the assumption that α is never selected. □

Efficient ways of enforcing **C3** are based on the specific search strategy that is used to generate the reduced state space. For depth first search, we can use the fact that every cycle includes an edge that goes back to a node on the search stack. Such an edge is also called a *back edge*. Thus, we strengthen **C3** in the following manner.

C3′ If s is not fully expanded, then no transition in $ample(s)$ may reach a state that is on the search stack.

We thus always try to select an ample set that does not include a back edge. If we do not succeed, the current state is fully expanded.

In breadth first search, the search progresses in levels, where level k consists of a set of states reachable from the initial states using k transitions. A necessary condition for closing a cycle during breadth first search is the following: A transition applied to a state s in the current level results in a state in the current or previous level of the breadth first search. This condition is not sufficient. Consequently, using this condition to detect when a cycle is closed may cause more states than necessary to be fully expanded.

10.5.2 Heuristics for Ample Sets

In view of the complexity results in Section 10.5.1 we give some heuristics for calculating ample sets. The algorithm will depend on the model of computation. We will consider shared variables and message passing with handshaking and with queues.

Common to all of these models of computation is the notion of a *program counter*, which is part of the state. We will denote the program counter of a process P_i in a state s by $pc_i(s)$.

In order to present the algorithm, we will use the following notation:

- $pre(\alpha)$ is a set of transitions that includes the transitions whose execution may enable α. More formally, $pre(\alpha)$ includes all the transitions β such that there exists a state s for which $\alpha \notin enabled(s)$, $\beta \in enabled(s)$, and $\alpha \in enabled(\beta(s))$.
- $dep(\alpha)$ is the set of transitions that are dependent on α, that is,

$\{\beta | (\beta, \alpha) \in D\}$.

- T_i is the set of transitions of process P_i. $T_i(s) = T_i \cap enabled(s)$ denotes the set of transitions of P_i that are enabled in the state s.
- $current_i(s)$ is the set of transitions of P_i that are enabled in some state s' such that $pc_i(s') = pc_i(s)$. The set $current_i(s)$ always contains $T_i(s)$. In addition, it may include transitions whose program counter has the value $pc_i(s)$, but are not enabled in s.

Note that on any path starting from s, some transition in $current_i(s)$ must be executed before other transitions of T_i can execute. The definitions of $pre(\alpha)$ and the dependency relation D (which directly effects $dep(\alpha)$) may not be exact. The set $pre(\alpha)$ may contain transitions that do not enable α. Likewise, the dependency relation D may also include pairs of transitions that are actually independent. This freedom makes it possible to calculate ample sets efficiently while still preserving the correctness of the reduction.

The above definitions are extended to sets in the natural way. For instance, $dep(T) = \cup_{\alpha \in T} dep(\alpha)$.

Next, we specialize $pre(\alpha)$ for various models of computation. Recall that $pre(\alpha)$ includes all transitions whose execution from some state can enable α. We construct $pre(\alpha)$ as follows:

- The set $pre(\alpha)$ includes the transitions of the processes that contain α and that can change the program counter to a value from which α can execute.
- If the enabling condition for α involves shared variables then $pre(\alpha)$ includes all other transitions that can change these shared variables.

- If α involves message passing with queues, that is, α sends or receives data on some queue q, then $pre(\alpha)$ includes the transitions of other processes that receive or send data, respectively, through q.

We now describe the dependency relation for the different models of computation.

1. Pairs of transitions that share a variable, which is changed by at least one of them, are dependent.

2. Pairs of transitions belonging to the same process are dependent. This includes in particular pairs of transitions in $current_i(s)$ for any given state s and process P_i. Note that a transition that involves handshaking or rendezvous communication as in CSP or ADA can be treated as a joint transition of both processes. Therefore, it depends on all of the transitions of both processes.

3. Two send transitions that use the same message queue are dependent. This is because executing one may cause the message queue to fill, disabling the other. Also, the contents of the queue depends on their order of execution. Similarly, two receive transitions are dependent.

Note that a pair of send and receive transitions in different processes, which use the same message queue are independent. This is because any one of these transitions can potentially enable the other but can not disable it.

An obvious candidate for $ample(s)$ is the set $T_i(s)$ of transitions enabled in s for some process P_i. Because the transitions in $T_i(s)$ are interdependent, an ample set for s must include either all of the transitions or none of them. To construct an ample set for the current state s, we start with some process P_i such that $T_i(s) \neq \emptyset$. We want to check whether $ample(s) = T_i(s)$ satisfies Condition **C1**. There are two cases in which this selection might violate **C1**. In both of these cases, some transitions independent of those in $T_i(s)$ are executed, eventually enabling a transition α that is dependent on $T_i(s)$. The independent transitions in the sequence cannot be in T_i, since all the transitions of P_i are interdependent.

1. In the first case, α belongs to some other process P_j. A necessary condition for this to happen is that $dep(T_i(s))$ includes a transition of process P_j. By examining the dependency relation, this condition can be checked effectively.

2. In the second case, α belongs to P_i. Suppose that the transition $\alpha \in T_i$ which violates **C1** is executed from a state s'. The transitions executed on the path from s to s' are independent of $T_i(s)$ and hence, are from other processes. Therefore, $pc_i(s') = pc_i(s)$. So α must be in $current_i(s)$. In addition, $\alpha \notin T_i(s)$, otherwise it does not violate **C1**. Thus, $\alpha \in current_i(s) \setminus T_i(s)$.

Since α is not in $T_i(s)$, it is disabled in s. Therefore, a transition in $pre(\alpha)$ must be included in the sequence from s to s'. A necessary condition for this case is that $pre(current_i(s) \setminus T_i(s))$ includes transitions of processes other than P_i. This condition can also be checked effectively.

In both cases we discard $T_i(s)$ as an ample set, and can try the transitions $T_j(s)$ of another process j as a candidate for $ample(s)$. Note that we take a conservative approach discarding some ample sets even though at run-time it might be that Condition **C1** would actually not be violated.

The following code checks Condition **C1** for the enabled transitions of a process P_i, as explained above.

```
function check_C1(s, P_i)
    for all P_j ≠ P_i do
        if dep(T_i(s)) ∩ T_j ≠ ∅
                    or pre(current_i(s) \ T_i(s)) ∩ T_j ≠ ∅ then
            return False;
        end if;
    end for all;
    return True;
end function
```

The function $check\_C2$ is given a set of transitions and returns *True* if all of the transitions in the set are invisible. Otherwise, it returns *False*.

```
function check_C2(X)
    for all α ∈ X do
        if visible(α) then return False;
    return True;
end function
```

The procedure $check\_C3'$ tests whether the execution of a transition in a given set $X \subseteq enabled(s)$ is still on the search stack. For that, we can use our marking of the states as *on_stack* or *completed* in Figure 10.2. Recall that a state is *on_stack* when the state is on the search stack.

```
function check_C3'(s, X)
    for all α ∈ X do
        if on_stack(α(s)) then return False;
    return True;
```



```
end function
```

The algorithm for $ample(s)$ tries to find a process P_i such that $T_i(s)$ satisfies all the conditions **C0** to **C3**. If no such process can be found, *ample* returns the set $enabled(s)$.

```
function ample(s)
    for all P_i such that T_i(s) ≠ ∅ do
        if check_C1(s, P_i) and check_C2(T_i(s))
                    and check_C3'(s, T_i(s)) then
            return T_i(s);
        end if;
    end for all;
    return enabled(s);
end function
```

The SPIN [138, 140] system includes an implementation [139] of the partial order reduction. The heuristics used for selecting ample sets are similar to the ones described in this section. However, in SPIN, for many of the states, Conditions **C0**, **C1**, and **C2** are precomputed when the system being verified is translated into its internal representation.

10.5.3 On-the-Fly Reduction

In previous sections of this chapter, the model-checking algorithm was explained as a two-phase process. The reduced state-space is constructed in the first phase. In the second phase, an LTL model-checking algorithm is used to check the correctness of a formula in the reduced state graph. In practice, many model checkers work in a more efficient manner. They combine the construction of the state graph with checking that it satisfies the specification. As shown in Section 9.5, it is frequently possible to identify on-the-fly that the system violates the specification before completing the construction of the state graph. The partial order reduction can be used in conjunction with on-the-fly model checking.

The only condition that needs special attention is the cycle closing Condition **C3**. The cycles in the product of the state graph and the property automaton are not necessarily the same as the ones in the reduced state graph generated in the off-line algorithm. To see this, observe that each state $\langle s, q \rangle$ in the product is a pair of a system state s and a state q of the property automaton. Assume that a cycle is closed at state s in the state graph. In the product, the state s may be paired with a different component of the automaton when it is encountered the second time. Thus, it cannot close a cycle. However, it can be shown [209] that it is correct to check Condition **C3'** with respect to cycles of the product. Intuitively, the purpose of **C3'** is to avoid postponing the inclusion of some transitions forever in the reduced graph. This is still guaranteed when **C3'** is applied to the cycles of the product. A formal proof appears in [209].

A subtle point arises when the double DFS procedure described in Section 9.3 is used with the partial order reduction. In this case, the order in which the graph is traversed may differ in the first and second phases of the search. As a consequence cycles may be closed at different states in the two phases. Thus, some additional information must be propagated between the two phases, to ensure that the same ample sets will be chosen in both [141].

10.6 Correctness of the Algorithm

Let M be the full state graph of some system. Let M' be a reduced state graph constructed using the partial order reduction algorithm described in Section 10.1.

A *string* is a sequence of transitions from T. Let T^* be the set of all the strings over T. Denote by $vis(v)$, where v is either finite or infinite string, the projection of v onto the visible transitions. Thus, if a and b are visible and c and d are not, then $vis(abddbcbaac) = abbbaa$. Let $tr(\sigma)$ be the sequence of transitions on a path σ. Let v, w be two finite strings. We write $v \sqsubset w$ if v can be obtained from w by erasing one or more transitions. For example $abbcd \sqsubset aabcbccde$. We denote $v \sqsubseteq w$ if either $v = w$ or $v \sqsubset w$.

Let $\sigma \circ \eta$ denote the concatenation of the paths σ and η of M, where σ is finite, and the last state $last(\sigma)$ of σ is the same as the first state $first(\eta)$ of η. The *length* of a path σ, denoted $|\sigma|$, is the number of edges of σ.

Let σ be some infinite path of the full state graph M, starting with some initial state. We will construct an infinite sequence of paths $\pi_0, \pi_1, \ldots$, where $\pi_0 = \sigma$. Each path π_i will be decomposed into $\eta_i \circ \theta_i$, where η_i is of length i. Assuming that we have constructed the paths $\pi_0, \ldots, \pi_i$, we describe how to construct $\pi_{i+1} = \eta_{i+1} \circ \theta_{i+1}$. Let $s_0 = last(\eta_i) = first(\theta_i)$ and α the transition labeling the first edge of θ_i. Denote

$$\theta_i = s_0 \xrightarrow{\alpha_0=\alpha} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots.$$

There are two cases:

A. $\alpha \in ample(s_0)$. Then select $\eta_{i+1} = \eta_i \circ (s_0 \xrightarrow{\alpha} \alpha(s_0))$. θ_{i+1} is $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$, that is, θ_i without its first edge.

B. $\alpha \notin ample(s_0)$. By **C2**, all of the transitions in $ample(s_0)$ must be invisible since s_0 is not fully expanded. Here again, there are two cases, **B1** and **B2**:

B1. Some $\beta \in ample(s_0)$ appears on θ_i after some sequence of independent transitions $\alpha_0\alpha_1\alpha_2 \ldots \alpha_{k-1}$, that is, $\beta = \alpha_k$. Then there is a path $\xi = s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0=\alpha} \beta(s_1) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-1}} \beta(s_k) \xrightarrow{\alpha_{k+1}} s_{k+2} \xrightarrow{\alpha_{k+2}} \ldots$ in M. That is, β is moved to appear before $\alpha_0\alpha_1\alpha_2 \ldots \alpha_{k-1}$. Note that $\beta(s_k) = s_{k+1}$. Therefore, $\beta(s_k) \xrightarrow{\alpha_{k+1}} s_{k+2}$ is the same as $s_{k+1} \xrightarrow{\alpha_{k+1}} s_{k+2}$.

B2. Some $\beta \in ample(s_0)$ is independent of all the transitions that appear on θ_i. Then there is a path $\xi = s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0=\alpha} \beta(s_1) \xrightarrow{\alpha_1} \beta(s_2) \xrightarrow{\alpha_2} \ldots$ in M. That is, β is executed from s_0 and then applied to each state of θ_i.

In both cases $\eta_{i+1} = \eta_i \circ (s_0 \xrightarrow{\beta} \beta(s_0))$ and θ_{i+1} is the path that is obtained from ξ by removing the first transition $s_0 \xrightarrow{\beta} \beta(s_0)$.

Let η be the path such that the prefix of length i is η_i. The path η is well defined in that η_i is constructed from η_{i-1} by appending a single transition.

LEMMA 26 The following hold for all i, j such that $j \geq i \geq 0$.

1. $\pi_i \sim_{st} \pi_j$.
2. $vis(tr(\pi_i)) = vis(tr(\pi_j))$.
3. Let ξ_i be a prefix of π_i and ξ_j be a prefix of π_j such that $vis(tr(\xi_i)) = vis(tr(\xi_j))$. Then $L(last(\xi_i)) = L(last(\xi_j))$.

Proof It is sufficient to consider the case where $j = i + 1$. Consider the three ways of constructing π_{i+1} from π_i. In case **A**, $\pi_i = \pi_{i+1}$, and all three parts of the lemma hold trivially.

Next, consider case **B1** of the construction, in which π_{i+1} is obtained from π_i by executing some invisible transition β in π_{i+1} earlier than it is executed in π_i. In this case, we replace the sequence $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-2}} s_{k-1} \xrightarrow{\beta} s_k$ by $s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0} \beta(s_1) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-2}} \beta(s_{k-1})$. Because β is invisible, corresponding states have the same label, that is, for each $0 < l \leq k$, $L(s_l) = L(\beta(s_l))$. Also, the order of the visible transitions remains unchanged. Parts 1, 2, and 3 follow immediately.

Finally, in case **B2** of the construction, the difference between π_i and π_{i+1} is that π_{i+1} includes an additional invisible transition β. Thus, we replace some suffix $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ of π_i by $s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0=\alpha} \beta(s_1) \xrightarrow{\alpha_1} \beta(s_2) \xrightarrow{\alpha_2} \ldots$. So, $L(s_l) = L(\beta(s_l))$ for $l \geq 0$. Again, the order of the visible transitions remains unchanged. As in the previous case, parts 1, 2, and 3 follow immediately. □

LEMMA 27 Let η be the path constructed as the limit of the finite paths η_i. Then, η belongs to the reduced state graph M'.

Proof By induction on the length of the prefixes η_i of η. The base case is that η_0 is a single node, which is an initial state in S. According to the reduction algorithms, all the initial states are included in S' as well. For the inductive step, assume that η_i is in M'. Then notice that η_{i+1} is obtained from η_i by appending a transition from $ample(last(\eta_i))$. □

The following three lemmas will be used to show that the path η that is constructed as the limit of the finite paths η_i contains all of the visible transitions of σ, and in the same order.

LEMMA 28 Let α be the first transition on θ_i. Then there exists $j > i$ such that α is the last transition of η_j, and for $i \leq k < j$, α is the first transition of θ_k.

Proof According to the above construction, if α is the first transition of θ_k, then either it is the first transition of θ_{k+1} (case **B**), or it will become the last transition of η_{k+1} (case **A**). We need to show that the first case cannot hold for every $k \geq i$. Suppose, on the contrary, that this is the case. Let $s_k = first(\theta_k)$. Consider the infinite sequence $s_i, s_{i+1}, \ldots$. According to the above construction, $s_{k+1} = \gamma_k(s_k)$ for some $\gamma_k \in ample(s_k)$. Moreover, because α is the first transition of θ_k and was not selected in case **A** to be moved to η_{k+1}, α must be in $enabled(s_k) \setminus ample(s_k)$. Because the number of states in S is finite, there is some state s_k that is the first to repeat on the sequence $s_i, s_{i+1}, \ldots$. Thus, there is a cycle $s_k, s_{k+1}, \ldots s_r$, with $s_r = s_k$, where α does not appear in any of the ample sets. This violates Condition **C3**. □

LEMMA 29 Let γ be the first visible transition on θ_i and $prefix_\gamma(\theta_i)$ be the maximal prefix of $tr(\theta_i)$ that does not contain γ. Then one of the following holds:

- γ is the first transition of θ_i and the last transition of η_{i+1}, or
- γ is the first visible transition of θ_{i+1}, the last transition of η_{i+1} is invisible, and $prefix_\gamma(\theta_{i+1}) \sqsubseteq prefix_\gamma(\theta_i)$.

Proof The first case of the lemma holds when γ is selected from $ample(s_i)$ and becomes the last transition of η_{i+1}, according to case **A** of the construction. If this does not happen, there exists another transition β that is appended to η_i to form η_{i+1}. The transition β cannot be visible. Otherwise, according to Condition **C2**, $ample(s_i) = enabled(s_i)$. By case **B1** of the construction, β must be the first transition of θ_i. But then β is a visible transition that precedes γ in θ_i, a contradiction.

There are three possibilities:

1. β appears on θ_i before γ (case **B1** in the construction),
2. β appears on θ_i after γ (case **B1** in the construction), or
3. β is independent of all the transitions of θ_i (case **B2** in the construction).

According to the above construction, in (1), $prefix_\gamma(\theta_{i+1}) \sqsubset prefix_\gamma(\theta_i)$ since β is removed from the prefix of θ_i before γ when constructing θ_{i+1}. In (2) and (3), $prefix_\gamma(\theta_{i+1}) = prefix_\gamma(\theta_i)$ since the prefix of θ_{i+1} that precedes the transition γ has the same transitions as the corresponding prefix of θ_i. □

LEMMA 30 Let v be a prefix of $vis(tr(\sigma))$. Then there exists a path η_i such that $v = vis(tr(\eta_i))$.

Proof By induction on the length of v. The base holds trivially for $|v| = 0$. In the inductive step we must prove that if $v\gamma$ is a prefix of $vis(tr(\sigma))$ and there is a path η_i such that $vis(tr(\eta_i)) = v$, then there is a path η_j with $j > i$ such that $vis(tr(\eta_j)) = v\gamma$. Thus, we need to show that γ will be eventually added to η_j for some $j > i$, and that no other visible transition will be added to η_k for $i < k < j$. According to case **A** in the construction, we may add a visible transition to the end of η_k to form η_{k+1} only if it appears as the first transition of θ_k. Lemma 29 shows that γ remains the first visible transition in successive paths θ_k after θ_i unless it is being added to some η_j. Moreover, the sequence of transitions before γ can only shrink. Lemma 28 shows that the first transition in each θ_k is eventually removed and added to the end of some η_l for $l > k$. Thus, γ as well is eventually added to some sequence η_j. □

THEOREM 12 The structures M and M' are stuttering equivalent.

Proof Each infinite path of M' that begins from an initial state must also be a path of M, for it is constructed by repeatedly applying transitions from the initial state. We need to show that for each path $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ in M, where s_0 is an initial state, there exists a path $\eta = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$ in M' such that $\sigma \sim_{st} \eta$. We will show that the path η that is constructed above for σ is indeed stuttering equivalent to σ.

First, we show that σ and η have the same sequence of visible transitions, that is, $vis(tr(\sigma)) = vis(tr(\eta))$. According to Lemma 30, η contains the visible transitions of σ in the same order, because for any prefix of σ with m visible transitions, there is a prefix η_i of η with the same m visible transitions. On the other hand, σ must contain the visible transitions of η in the same order. Take any prefix η_i of η. According to Lemma 26, $\pi_i = \eta_i \circ \theta_i$ has the same visible transitions as $\pi_0 = \sigma$. Thus, σ has a prefix with the same sequence of visible transitions as η_i.

We construct two infinite sequences of indexes $0 = i_0 < i_1 < \ldots$ and $0 = j_0 < j_1 < \ldots$ that define corresponding stuttering blocks of σ and η, as required in the definition of stuttering. Assume that both $\sigma = \pi_0$ and η have at least n visible transitions. Let i_n be the length of the smallest prefix ξ_{i_n} of σ that contains exactly n visible transitions. Let j_n be the length of the smallest prefix η_{j_n} of η that contains the same sequence of visible transitions as ξ_{i_n}. Recall that η_{j_n} is a prefix of π_{j_n}. Then by part 3 of lemma 26, $L(s_{i_n}) = L(r_{j_n})$. By the definition of visible transitions we also know that if $n > 0$, for $i_{n-1} \leq k < i_n - 1$, $L(s_k) = L(s_{i_{n-1}})$. This is because i_{n-1} is the length of the smallest prefix $\xi_{i_{n-1}}$ of σ that contains exactly $n - 1$ visible transitions. Thus, there is no visible transition between i_{n-1} and $i_n - 1$. Similarly, for $j_{n-1} \leq l < j_n - 1$, $L(r_l) = L(r_{j_{n-1}})$.

If both σ and η have infinitely many visible transitions, then this process will construct two infinite sequences of indexes. In the case where σ and η contain only a finite number of visible transitions m, we have that for $k > i_m$, $L(s_k) = L(s_{i_m})$ and for $l > j_m$, $L(r_l) = L(r_{j_m})$. We then set for $k \geq m$, $i_{k+1} = i_k + 1$ and $j_{k+1} = j_k + 1$. By the above, for $k \geq 0$, the blocks of states $s_{i_k}, s_{i_k+1}, \ldots, s_{i_{k+1}-1}$ and $r_{j_k}, r_{j_k+1}, \ldots, r_{j_{k+1}-1}$ are corresponding stuttering blocks that have the same labeling. Thus, $\sigma \sim_{st} \eta$. □

10.7 Partial Order Reduction in SPIN

SPIN [138, 140] is an on-the-fly LTL model checker that uses explicit state enumeration and the partial order reduction. It was developed at Bell Laboratories by Gerard Holzmann and Doron Peled. The tool is used primarily for verifying asynchronous software systems, in particular communication protocols. It can check a model of a program for deadlocks or unreachable code or determine if it satisfies an LTL specification, based on the translation algorithm [124] described in Section 9.4. The tool uses the partial order reduction [139, 209] to limit the state space that is searched.

The input language for SPIN, called PROMELA, was developed by Gerard Holzmann. This language uses syntactic constructs from several different programming languages. PROMELA expressions are inherited from the language C [154]. Thus, the language has the operators '==' (equals), '! =' (not equals), '||' (logical or), '&&' (logical and), and '%' (reminder modulo an integer). Assignment is denoted by a single '=' symbol. Negation is denoted by prefixing a boolean expression by the operator '!'.

The syntax for communication commands is inherited from CSP [137]. Sending a message that contains the tag `tg` and the values $val_1, val_2, \ldots, val_n$ over channel `ch` is denoted by

`ch!tg`$(val_1, val_2, \ldots, val_n)$

in the sending process. Receiving a message with tag `tg` over channel `ch` is denoted by

`ch?tg`$(var_1, var_2, \ldots, var_n)$

in the receiving process. The message consists of n values that are stored in the variables $var_1, var_2, \ldots, var_n$. SPIN also allows untagged message passing. The language implements both message passing with queues and message passing using handshaking. In message passing with queues, a channel of some fixed length temporarily stores the values sent, so that the sending process can proceed to its next command, even if the receiving process is not yet ready to process the incoming data. In message passing with handshaking, a channel is defined in SPIN to be of length 0. Then, a send and a receive command

if	**do**
$:: guard_1 -> S_1$	$:: guard_1 -> S_1$
$:: guard_2 -> S_2$	$:: guard_2 -> S_2$
⋮	⋮
$:: guard_n -> S_n$	$:: guard_n -> S_n$
fi	**od**

Figure 10.10
Conditionals and loops in SPIN.

with the same channel and tag (if a tag is present) are executed simultaneously. This results in the assignment of val_i to var_i, for $1 \leq i \leq n$.

The conditional constructs and loops are based on Dijkstra's *Guarded Commands* [95] and use the syntax in Figure 10.10.

Each guard consists of a condition, a communication command, or both. In order for a guard to be *passable*, its condition must hold, and its communication command must not be blocked. In message passing with queues, a send command is blocked when the queue is full, and a receive command is blocked when the queue is empty. In message passing based on handshaking, communication is blocked when only one of the communicating processes is ready to send or to receive.

When executing the `if` construct and at each iteration of the `do` loop, one of the passable guards $guard_i$ is selected nondeterministically and then the corresponding command S_i is executed. A `do` loop repeats until either a *goto* command forces a branch to a particular label outside its scope, or a `break` command forces a skip to the first command after the `do` loop.

The reduction obtained by using the ample set technique described in Section 10.3 is demonstrated using the *leader election* algorithm developed by Dolev, Klawe, and Rodeh [102]. This algorithm operates on a ring of N processes. Each process initially has a unique number. The purpose of this algorithm is to find the largest number assigned to a process. The ring of processes is unidirectional; hence, each process can receive messages from its left and send messages to its right.

Initially, each process P_i is *active* and holds some integer value in its local variable `my_val`. As long as P_i is active, it is *responsible* for some value. This value may change during the execution of the algorithm. The current value of P_i is held in the variable `max`. A process becomes *passive* when it finds out that it does not hold a value that can be the maximum one. A passive process can only pass messages from left to right. Each active process P_i sends its own value to the right and then waits to receive the value of the closest active process P_j on its left. This value is received using a communication command tagged with `one`.

If the value received by P_i is the same as the value it sent, then P_i can conclude that it is the only active process and, hence, its value is the maximum. Then process P_i sends this value to the right with the tag `winner`. Every other process receives this value and sends it to the right exactly once, so that all the processes can learn the winning number.

If the value received by P_i is not the same as the value it sent, then P_i waits for a second message, tagged with `two`, that includes the value of the second closest active process on its left P_k. Then, P_i compares its own value with the two values it received from P_j and P_k. If the value received from P_j is the greatest among the three, then P_i keeps this value. That is, P_i becomes responsible for the role of the closest active process P_j. Otherwise, P_i becomes passive.

The execution of the algorithm can be divided into phases. In each *phase*, except the last, all of the active processes receive messages tagged with `one` and `two`. In the last phase, the surviving process receives its own value via a message tagged with `one` and then this value is propagated around the ring.

The protocol guarantees low message complexity $O(N \times log(N))$. This complexity bound holds because at least half of the active processes become passive in each phase. To see this, consider the case where P_i remains active. Then the value of P_j must be bigger than the values of P_i and P_k. If P_j also survives, then the value of P_k must be larger than the value of P_j. This is a contradiction. Thus, in each phase except for the last, if a process remains active, the first active process to its left must become passive. In each phase, the number of messages passed is limited to $2 \times N$, since each process receives two messages from its left neighbor.

The PROMELA code for the leader election algorithm appears in Figure 10.11. We omit the code for initializing the processes. This includes assigning a distinct number to each process and starting the execution of that process. The channel $q[(i+1)\%N]$ is used to send messages from process P_i to process $P_{i+1\%N}$, where $\%N$ denotes the reminder modulo N.

The property that we checked is given by the LTL formula

noLeader **U G** *oneLeader*.

This formula asserts that in each execution there is no leader until some time in the future when a leader is selected. From that point onward, there is exactly one leader. The predicates *noLeader* and *oneLeader* are defined as `number_leaders` == 0 and `number_leaders` == 1, respectively.


```
#define noLeader        (number_leaders == 0)
#define oneLeader       (number_leaders == 1)

byte number_leaders = 0;

#define N       6       /* number of processes in the ring */
#define L       12       /* 2xN */
byte I;

mtype = { one, two, winner };
chan q[N] = [L] of { mtype, byte};

proctype P (chan in, out; byte my_val)
{       bit Active = 1, know_winner = 0;
        byte number, max = my_val, neighbor;

        out!one(my_val);
        do
        :: in?one(number) ->  /*Get left active neighbor value*/
           if
           :: Active ->
              if
              :: number != max ->
                 out!two(number); neighbor = number
              :: else ->
                 know_winner = 1; out!winner(number);
              fi
           :: else ->
              out!one(number)
           fi
```

Figure 10.11
The leader election protocol in PROMELA.

The negation of the checked property is automatically translated into a Büchi automaton, based on the algorithm described in Section 9.4. An additional minimization stage combines nodes with the same branching structure. The automaton is described using a special syntactical construct of PROMELA called the *never claim*. The reason for this name is that the automaton, obtained by translating the negation of the checked property, repre-

```
:: in?two(number) ->  /*Get second left active neighbor value*/
   if
   :: Active ->
      if
      :: neighbor > number && neighbor > max ->
         max = neighbor; out!one(neighbor)
      :: else ->
         Active = 0 /* Becomes passive */
      fi
   :: else ->
      out!two(number)
   fi

:: in?winner(number) ->
   if
   :: know_winner
   :: else -> out!winner(number)
   fi;
   break
od
```

Figure 10.11 (continued)

sents the computations that should never happen. The never claim for the above property is shown in Figure 10.12. The label of each initial node contains the word `init` and the label of each accepting node contains the word `accept`.

SPIN intersects the automaton extracted from the program and the never claim automaton. This intersection is done on-the-fly, using the double-DFS algorithm presented in Section 9.3 and the partial order reduction. If the intersection is not empty, an error trace is reported.

The experimental results are summarized in the table in Figure 10.13. The experiments were conducted on an SGI *Challenge* machine. The memory in the table is given in megabytes. Verifying the algorithm with five and six processes without using the partial order reduction did not terminate. The table indicates that the case of five processes without partial order reduction was still running after forty hours. The results of this experiment clearly demonstrates how the partial order reduction is able to alleviate the state explosion problem.

```
never {    /* !(noLeader U [] oneLeader) */
T0_init:
   if
   :: (! ((noLeader))) -> goto T0_S28
   :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
   :: (1) -> goto T0_S9
   :: (! ((oneLeader))) -> goto accept_S1
   fi;
accept_S1:
   if
   :: (! ((noLeader))) -> goto T0_S28
   :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
   :: (1) -> goto T0_S9
   :: (! ((oneLeader))) -> goto T0_init
   fi;
accept_S9:
   if
   :: (! ((noLeader))) -> goto T0_S28
   :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
   :: (1) -> goto T0_S9
   :: (! ((oneLeader))) -> goto T0_init
   fi;
accept_S28:
   if
   :: (1) -> goto T0_S28
   :: (! ((oneLeader))) -> goto accept_all
   fi;
T0_S9:
   if
   :: (! ((noLeader))) -> goto T0_S28
   :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_S28
   :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
   :: (! ((oneLeader))) -> goto accept_S9
   :: (1) -> goto T0_S9
   :: (! ((oneLeader))) -> goto accept_S1
   fi;
T0_S28:
   if
   :: (1) -> goto T0_S28
   :: (! ((oneLeader))) -> goto accept_all
   fi;
accept_all:
   skip
}
```

Figure 10.12
The never claim for the specification.

Procs:	Non-reduced States	Non-reduced Memory	Non-reduced Time	Reduced States	Reduced Memory	Reduced Time
3	15929	1.801	13.8 sec	1435	1.493	0.6 sec
4	522255	15.727	9.3 min	8475	1.698	3.5 sec
5		>128	>40 hours	57555	3.234	28.7 sec
6				434083	15.625	4.1 min

Figure 10.13
Experimental results for the partial order reduction.

11 Equivalences and Preorders between Structures

In this chapter we will show how to avoid the state explosion problem by developing techniques that replace a large structure by a smaller structure which satisfies the same properties. We have already seen one example of this technique in Chapter 10 where the partial order reduction was used to reduce the size of structures while preserving the truth of LTL formulas that do not involve the next-time operator. More generally, given a logic $\mathcal{L}$ and a structure M, we would like to find a smaller structure M' that satisfies exactly the same set of formulas of the logic $\mathcal{L}$ as M. In order to accomplish this goal, we need a notion of equivalence between structures that can be efficiently computed and guarantees that two structures satisfy the same set of formulas in $\mathcal{L}$. We first consider the logic CTL* and *bisimulation equivalence* [207].

It is convenient to include a set of initial states S_0 and a set of atomic propositions AP with every structure M. Thus, a typical structure is $M = (AP, S, R, S_0, L)$. If fairness is also considered, then $M = (AP, S, R, S_0, L, F)$. Sometimes it is necessary to transform a structure that does not have fairness assumptions into one that does, while preserving the set of paths considered as computations. This can be accomplished by letting $F = \{S\}$.

Let $M = (AP, S, R, S_0, L)$ and $M' = (AP, S', R', S'_0, L')$ be two structures with the same set of atomic propositions AP. A relation $B \subseteq S \times S'$ is a *bisimulation relation* between M and M' if and only if for all s and s', if $B(s, s')$ then the following conditions hold:

1. $L(s) = L'(s')$.
2. For every state s_1 such that $R(s, s_1)$ there is s'_1 such that $R'(s', s'_1)$ and $B(s_1, s'_1)$.
3. For every state s'_1 such that $R'(s', s'_1)$ there is s_1 such that $R(s, s_1)$ and $B(s_1, s'_1)$.

The structures M and M' are *bisimulation equivalent* (denoted $M \equiv M'$) if there exists a bisimulation relation B such that for every initial state $s_0 \in S_0$ in M there is an initial state $s'_0 \in S'_0$ in M' such that $B(s_0, s'_0)$. In addition, for every initial state $s'_0 \in S'_0$ in M' there is an initial state $s_0 \in S_0$ in M such that $B(s_0, s'_0)$.

Figures 11.1 and 11.2 demonstrate simple examples of bisimulation equivalent structures. The figures show that unwinding a structure or duplicating some part of a structure may result in a bisimulation equivalent structure. Figure 11.3, on the other hand, shows two structures that are not bisimulation equivalent. In order to see this, note that the state labeled with b in M' does not correspond to any of the states labeled with b in M because none of these states have both a successor labeled by c and a successor labeled by d.

The following lemma is important in establishing the connection between CTL* and bisimulation equivalence. We say that two paths $\pi = s_0 s_1, \ldots$ in M and $\pi' = s'_0 s'_1, \ldots$ in M' *correspond* if and only if for every $i \geq 0$, $B(s_i, s'_i)$.

References

[1] S. Aggarwal, R. P. Kurshan, and K. Sabnani. A calculus for protocol specification and validation. In H. Rudin and C. H. West, eds., *Protocol Specification, Testing and Verification*, pp. 19–34. North Holland, 1983.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[3] R. Alur. Timed automata. NATO ASI Summer School on Verification of Digital and Hybrid Systems, 1998. (Available at www.cis.upenn.edu/alur/Nato97.ps.gz.)

[4] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qudeer, and S. K. Rajamani. Partial–order reduction in symbolic state space explosion. In O. Grumberg, ed., *9th International Conference on Computer Aided Verification, LNCS* 1254, pp. 340–351, Springer, 1997.

[5] R. Alur and D. L. Dill. Automata-theoretic verification of real-time systems. In Constance Heitmeyer and Dino Mandrioli, eds., *Formal Methods for Real-Time Computing*, pp. 55–80. Wiley, 1996.

[6] R. Alur and T. A. Henzinger, eds. *Proceedings of the 1996 Workshop on Computer-Aided Verification, LNCS* 1102. Springer, 1996.

[7] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

[8] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pp. 414–425. IEEE Computer Society Press, 1990.

[9] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science* 126(2): 183–235.

[10] R. Alur and R. P. Kurshan. Timing analysis in COSPAN. In R. Alur, T. A. Henzinger, and E. D. Sontag, eds., *Hybrid Systems III: Verification and Control, LNCS* 1066, pp. 220–231. Springer, 1995

[11] H. R. Andersen. Model checking and boolean graphs. In B. Krieg-Bruckner, ed., *Proceedings of the Fourth European Symposium on Programming, LNCS* 582, pp. 1–19. Springer, 1992.

[12] K. Apt and D. Kozen. Limits for automatic verification of finite-state systems. *IPL* 15: 307–309.

[13] A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Equivalences for fair Kripke structures. In *Proceedings of the 21st International Colloquium on Automata, Languages and Programming, LNCS* 820, pp. 364–375. Springer, 1994.

[14] A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In Wolper [249], pp. 155–166.

[15] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, eds., *24th International Colloquium on Automata, Languages, and Programming (ICALP '97), LNCS*, 1256, pp. 430–440. Springer, 1997.

[16] F. Balarin and A. Sangiovanni-Vincentelli. On the automatic computation of network invariants. In Dill [97], pp. 235–246.

[17] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In Courcoubetis [83], pp. 29–40.

[18] D. L. Beatty, R. E. Bryant, and C-J. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, 1991.

[19] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica* 20(1983): 207–226.

[20] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In von Bochmann and Probst [243], pp. 260–273.

[21] W. Bernard and P. Wolper. Partial-order methods for model checking: From linear time to branching time. In *Eleventh Annual IEEE Symposium on Logic in Computer Science*, pp. 294–303. IEEE Computer Society, 1996.

[22] O. Bernholtz, M. Y. Vardi, and P. Wolper. An automata theoretic approach to branching time model checking. In Dill [97], pp. 142–155.

[23] C. Berthet, O. Coudert, and J. C. Madre. New ideas on symbolic manipulations of finite state machines. In *IEEE International Conference on Computer Design*, 1990.

[24] N. Bjorner, et al. Step: The Stanford temporal prover—user's manual. Technical Report STAN-CS-TR-95-1562, Department of Computer Science, Stanford University, November 1995.

[25] G. von Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers* C-31(3).

[26] S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In L. Claesen, ed., *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[27] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* IEEE Computer Society Press, 1990.

[28] M. C. Browne and E. M. Clarke. SML: A high level language for the design and verification of finite state machines. In *IFIP WG 10.2 Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs*, pp. 269–292. IFIP, 1987.

[29] M. C. Browne, E. M. Clarke, and D. Dill. Checking the correctness of sequential circuits. In *Proceedings of the 1985 International Conference on Computer Design*, pp. 545–548. IEEE, 1985..

[30] M. C. Browne, E. M. Clarke, and D. L. Dill. Automatic circuit verification using temporal logic: Two new examples. In G. J. Milne and P. A. Subrahmanyam, eds., *Formal Aspects of VLSI Design*. Elsevier, 1986.

[31] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers* C-35(12): 1035–1044.

[32] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science* 59(1–2): 115–131.

[33] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite-state processes. *Information and Computation* 81(1): 13–31.

[34] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* C-35(8): 677–691.

[35] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers* 40(2): 205–213.

[36] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys* 24(3): 293–318.

[37] R. E. Bryant and C-J. Seger. Formal verification of digital circuits using symbolic ternary system models. In Clarke and Kurshan [71], pp. 33–43.

[38] J. A. Brzozowski and C. J. H. Seger. Advances in asynchronous circuit theory. Part II: Bounded inertial delay models, MOS circuits, design techniques. *Bulletin of the European Association for Theoretical Computer Science* 43(3): 199–263.

[39] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, pp. 1–11. Stanford University Press, 1960.

[40] J. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In Dill [97], pp. 68–81.

[41] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon University, 1992.

[42] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 403–407. IEEE, 1991.

[43] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, eds., *Proceedings of the 1991 International Conference on VLSI*, pp. 49–58. August 1991.

[44] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits* 13(4): 401–424.

[45] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 46–51. IEEE, 1990.

[46] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2): 142–170.

[47] M. Burrow, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Transactions on Computer Systems* 8(1): 18–36.

[48] R. M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress 74*, pp. 308–312. North Holland, 1974.

[49] S. V. Campos. *A Quantitative Approach to the Formal Verification of Real-Time System*. PhD thesis, Carnegie Mellon University, 1996.

[50] S. V. Campos and E. M. Clarke. Real-time symbolic model checking for discrete time models. In *First AMAST International Workshop in Real-Time Systems*. Springer, 1993.

[51] S. V. Campos, E. M. Clarke, W. Marrero, and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In *ACM Workshop on Languages Compilers and Tools for Real-Time Systems*, 1995.

[52] S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Real-Time Systems Symposium*. IEEE, 1994.

[53] S. V. Campos, E. M. Clarke, and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In *Proceedings of the IEEE International Conference on Computer Design*, pp. 73–79. IEEE, 1995.

[54] M. Chiodo, T. R. Shiple, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Automatic compositional minimization in CTL model checking. In ICCAD92 [146], pp. 172–178.

[55] C-T. Chou and D. Peled. Verifying a model-checking algorithm. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 241–257. Springer, 1996.

[56] L. Claesen, ed. *Proceedings of the 11th International Symposium on Computing Hardware Description Languages and their Applications*. North Holland, 1993.

[57] D. Clarke, H. Ben-Abdallah, I. Lee, H. Xie, and O. Sokolsky. XVERSA: an integrated graphical and textual toolset for the specification and analysis of resource-bound real-time systems. In *Computer-Aided Verification*, pp. 402–405. Springer, 1996.

[58] E. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design* 9: 77–104.

[59] E. M. Clarke and I. A. Draghicescu. Expressibility results for linear time and branching time logics. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, *LNCS* 354, pp. 428–437. Springer, 1988.

[60] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold and N. D. Jones, eds., *Proceedings of the 15th Colloquium on Trees in Algebra and Programming*, *LNCS* 407, pp. 103–116. Springer, 1990.

[61] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, *LNCS* 131. Springer, 1981.

[62] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Language*, January 1983.

[63] 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.

[64] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [83], pp. 450–462.

[65] E. M. Clarke, O. Grumberg, and H. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design* 10(1): 47–71.

[66] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In Claesen [56].

[67] E. M. Clarke, O. Grumberg, and S. Jha. Parametrized networks. In S. Smolka and I. Lee, eds., *Proceedings of the 6th International Conference on Concurrency Theory*, *LNCS* 962, pp. 395–407. Springer, 1995.

[68] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19(5): 726–750.

[69] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems* 16(5): 1512–1542.

[70] E. M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proceedings of the IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[71] E. M. Clarke and R. P. Kurshan, eds. *Workshop on Computer-Aided Verification. 2nd International Conference, CAV'90. Proceedings*, *LNCS* 531. Springer, 1990.

[72] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In J. A. Darringer and F. J. Rammig, eds., *Proceedings of the 9th International Symposium on Computer Hardware Description Languages and Their Applications*, pp. 281–295. North Holland, 1989.

[73] R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal mu-calculus. In v. Bochmann and Probst [243], pp. 410–422.

[74] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design* 2(2): 121–147.

[75] R. W. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica* 27: 725–747.

[76] R. W. Cleaveland, P. Lewis, S. Smolka, and O. Sokolsky. The concurrency factory: a development environment for concurrent systems. In Alur and Henzinger [6], pp. 398–401.

[77] R. W. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In Sifakis [231], pp. 24–37.

[78] R. W. Cleaveland and S. Sims. The NCSU concurrency workbench. In Alur and Henzinger [6], pp. 394–397.

[79] P. Clements, C. Heitmeyer, G. Labaw, and A. Rose. MT: a toolset for specifying and analyzing real-time systems. In *IEEE Real-Time Systems Symposium*, 1993.

[80] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1989.

[81] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Sifakis [231], pp. 365–373.

[82] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In Clarke and Kurshan [71], pp. 23–32.

[83] C. Courcoubetis, ed. *Proceedings of the 5th Workshop on Computer-Aided Verification*, June/July 1993.

[84] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design* 1: 275–288.

[85] C. Courcoubetis and M. Yannakakis. The complexity of probablistic verification. *Journal of the ACM* 42(4): 857–907.

[86] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedimgs of the 4th Annual ACM Symposium on Principles of Programming Language*, pp. 238–252, January 1977.

[87] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Language*, pp. 269–282, January 1979.

[88] D. Cyrluk and P. Narendran. Ground temporal logic: A logic for hardware verification. In Dill [97], pp. 247–259.

[89] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Technical University of Eindhoven, Eindhoven, 1995.

[90] D. Dams, R. Gerth, and O. Grumberg. Generation of reduced models for checking fragments of CTL. In *5th Conference on Computer-Aided Verification*, *LNCS* 697, pp. 479–490. Springer, 1993.

[91] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19(2): 253–291.

[92] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, *LNCS* 1066, pp. 208–219. Springer, 1996.

[93] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 16^{th} Real-Time Systems Symposium*, pp. 66–75. IEEE Computer Society Press, 1995.

[94] J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds. *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, *LNCS* 430. Springer,1989.

[95] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18(8): 453–457.

[96] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[97] D. L. Dill, ed. *Proceedings of the 1994 Workshop on Computer-Aided Verification*, *LNCS* 818. Springer, 1994.

[98] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings* Part E 133(5), 1986.

[99] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, ed., *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, *LNCS* 407, pp. 197–212. Springer, 1989.

[100] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In Wolper [249], pp. 54–69.

[101] P. Dixon. Multilevel cache architectures. Minutes of the Futurebus+ Working Group Meeting, December 1988.

[102] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms* 3: 245–260.

[103] E. A. Emerson. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs*. PhD thesis, Harvard University, 1981.

[104] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *LNCS* 85, *Automata, Languages and Programming*, pp. 169–181. Springer, 1980.

[105] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching time versus linear time. *Journal of the ACM* 33: 151–178.

[106] E. A. Emerson and C-L. Lei. Modalities for model checking: Branching time strikes back. *Twelfth Symposium on Principles of Programming Languages, New Orleans, La.*, pp. 84–96. ACM Press, January 1985.

[107] E. A. Emerson and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In LICS86 [174], pp. 267–278.

[108] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasen. Quantitative temporal reasoning. In Clarke and Kurshan [71], pp. 136–145.

[109] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pp. 85–94. ACM Press, 1995.

[110] E. A. Emerson and K. S. Namjoshi. Automated verification of parameterized synchronous systems. In Alur and Henzinger [6] pp. 87–98.

[111] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In Courcoubetis [83], pp. 463–478.

[112] E.A. Emerson, C.S. Jutla, and A.P. Sistla. On model-checking for fragments of mu-calculus. In Courcoubetis [83] pp. 385–396.

[113] J. C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In Alur and Henzinger [6], pp. 348–359.

[114] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, ed., *Mathematical Aspects of Computer Science*, pp. 19–32. American Mathematical Society, 1967.

[115] N. Francez. *The Analysis of Cyclic Programs*. PhD thesis, The Weizmann Institute of Science, 1976.

[116] N. Francez. *Fairness*. Springer, 1986.

[117] A. N. Fredette and R. W. Cleaveland. RTSL: a language for real-time schedulability analysis. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1993.

[118] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer Aided Design*. IEEE Computer Society Press, 1988.

[119] M. Fujita, H. Tanaka, and T. Moto-oka. Logic design assistance with temporal logic. In *Proceedings of the IFIP WG10.2 International Conference on Hardware Description Languages and their Applications*. 1985.

[120] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pp. 163–173. ACM, 1980.

[121] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[122] R. Gerber and I. Lee. A proof system for communicating shared resources. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1990.

[123] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM* 39: 675–735.

[124] R. Gerth, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pp. 3–18. Chapman & Hall, 1995.

[125] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd Workshop on Computer-Aided Verification*, *LNCS* 531, pp. 176–185. Springer, 1990.

[126] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proceedings of the 5th Conference on Computer-Aided Verification*, *LNCS*, 697, pp. 438–449. Springer, 1993.

[127] S. Graf. Verification of a distributed cache memory by using abstractions. In Dill [97], pp. 207–219.

[128] S. Graf and B. Steffen. Compositional minimization of finite state processes. In Clarke and Kurshan [71], pp. 186–196.

[129] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems* 16: 843–872.

[130] N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design* 11(2): 157–185.

[131] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering* 20(1): 13–28.

[132] R. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In Alur and Henzinger [6], pp. 423–427.

[133] Z. Har'El and R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal* 69(1): 45–59.

[134] C. Heitmeyer and D. Mandrioli. *Formal Methods for Real-Time Computing*. Wiley, 1996.

[135] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation* 3(2): 193–244.

[136] C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the Association for Computing Machinery* 12(10): 322–329.

[137] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[138] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[139] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Formal Description Techniques 1994*, pp. 197–211. Chapman & Hall, 1994.

[140] G. J. Holzmann and D. Peled. The state of spin. In *CAV'96: 8th International Conference on Computer Aided Verification*, *LNCS* 1102, pp. 385–389. Springer, 1996.

[141] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Second SPIN Workshop*, pp. 23–32. AMS, 1996.

[142] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[143] P. Huber, A. Jensen, L. Jepsen, and K. Jensen. Towards reachability trees for high-level Petri nets. In G. Rozenberg, ed., *Advances on Petri Nets*, pp. 215–233. 1984.

[145] G. E. Hughes and M. J. Creswell. *Introduction to Modal Logic*. Methuen, 1977.

[146] *IEEE Computer Society. 1992 Proceedings of the IEEE International Conference on Computer Aided Design*. IEEE Computer Society Press, 1992.

[147] IEEE Computer Society. *IEEE Standard for Futurebus+—Logical Protocol Specification*. IEEE Computer Socity Press, 1992. IEEE Standard 896.1–1991.

[148] C. W. Ip and D. L. Dill. Better verification through symmetry. In Claesen [56].

[149] D. Jackson. Abstract model checking of infinite specifications. In *Proceedings of Formal Methods Europe*, Barcelona, Oct. 1994.

[150] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pp. 321–332. North-Holland, 1983.

[151] B. Josko. Verifying the correctness of AADL-modules using model checking. In de Bakker et al. [94].

[152] J. J. Joyce and C-J. H. Seger. Linking BDD-based symbolic evaluation to interactive theorem proving. In *Proceedings of the 30th Design Automation Conference*. Association for Computing Machinery, 1993.

[153] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, *LNCS* 354, pp. 489–507. Springer, 1988.

[154] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[155] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional laguages. In O. Grumberg, ed., *9th International Conference on Computer Aided Verification (CAV'97)*, *LNCS* 1254, pp. 424–435. Springer, 1997.

[156] K. Keutzer. Hardware-software co-design and ESDA. In *Proceedings of the 31th Design Automation Conference*, pp. 435–436. June 1994.

[157] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science* 27: 333–354.

[158] F. Kröger. LAR: A logic of algorithmic reasoning. *Acta Informatica*, 8(3): .

[159] O. Kupferman and M. Y. Vardi. Verification of fair transition systems. In Alur and Henzinger [6], pp. 372–382.

[160] R. P. Kurshan. Analysis of discrete event coordination. In de Bakker et al. [94], pp. 414–453.

[162] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, pp. 170–172. Princeton University Press, 1994.

[163] R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In Courcoubetis [83], pp. 166–180.

[164] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS* 1384, pp. 345–357. Springer, 1998.

[165] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pp. 239–247. ACM, 1989.

[166] L. Lamport. "Sometimes" is sometimes "Not Never." In *Annual ACM Symposium on Principles of Programming Language*, pp. 174–185. ACM, 1980.

[167] L. Lamport. What good is temporal logic. In *Information Processing 83*, pp. 657–668. Elsevier, 1983.

[168] K. G. Larsen. Modal specifications. In Sifakis [231], pp. 232–246.

[169] K. G. Larsen. Efficient local correctness checking. In v. Bochmann and Probst [243], pp. 30–43.

[170] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and symbolic model-checking of real-time systems. In *Proceedings of the 16th Real-Time Systems Symposium*, pp. 76–87. IEEE Computer Society Press, 1995.

[171] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1990.

[172] J. P. Lehoczky, L. Sha, J. K. Strosnider, and H. Tokuda. Fixed priority scheduling theory for hard real-time systems. In *Foundations of Real-Time Computing—Scheduling and Resource Management*. Kluwer Academic, 1991.

[173] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Language*, pp. 97–107. ACM, 1985.

[174] *Proceedings of the 1st Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1986.

[175] B. Lin and A. R. Newton. Efficient symbolic manipulation of equvialence relations and classes. In *Proceedings of the 1991 International Workshop on Formal Methods in VLSI Design*, pp. 46–61. January 1991.

[176] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 20(1): 46–61.

[177] C. D. Locke, D. R. Vogel, and T. J. Mesler. Building a predictable avionics platform in Ada: a case study. In *IEEE Real-Time Systems Symposium*. IEEE Computer Socity Press, 1991.

[178] D. Long, A. Browne, E. Clarke, S. Jha, and W. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In Dill [97], pp. 338–350.

[179] D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon University, 1993.

[180] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS* 1055, pp. 147–166. Springer, 1996.

[181] E. Macii, B. Plessier, and F. Somenzi. Verification of systems containing counters. In ICCAD92 [146], pp. 179–182.

[182] S. MacLane and G. Birkhoff. *Algebra*. MacMillan, 1968.

[183] A. Mader. Tableau recycling. In v. Bochmann and Probst [243], pp. 330–342.

[184] Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. In H. T. Kung, B. Sproull, and G. Steele, eds., *VLSI Systems and Computations*. Computer Science Press, 1981.

[185] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincenteli. Logic verification using binary decision diagrams in a logic synthesis environment. In *International Conference on Computer-Aided Design*, pp. 6–9. 1988.

[186] Z. Manna and A. Pnueli. *Temporal Verifications of Reactive Systems—Safety*. Springer, 1995.

[187] R. Marelly and O. Grumberg. GORMEL—Grammar ORiented ModEL checker. Technical Report 697, The Technion, October 1991.

[188] W. Marrero, E. M. Clarke, and S. Jha. A model checker for authentication protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[189] A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs, ed., *Proceedings of the 1985 Chapel Hill Conference on VLSI*, 1985.

[190] K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. In v. Bochmann and Probst [243], pp. 164–177.

[191] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[192] K. L. McMillan and D.L. Dill. Algorithms for interface timing verification. In ICCAD92 [146], pp. 48–51.

[193] C. Meadows. A model of computation for the NRL protocol analyzer. In *Proceedings of the 1994 Computer Security Foundations Workshop*. IEEE Computer Society Press, 1994.

[194] J. Millen. The Interrogator model. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pp. 251–260. IEEE Computer Society Press, 1995.

[195] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pp. 481–489. 1971.

[196] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[197] B. Mishra and E.M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science* 38: 269–291.

[198] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering* SE-7, No. 4: 417–426.

[199] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1997.

[200] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*. Princeton University Press, 1956.

[201] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.

[202] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[203] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica* 18: 265–287.

[204] J. K. Ousterhout. A switch-level timing verifier for digital MOS VLSI. *IEEE Transactions on Computer-Aided Design* 4(3): 336–349.

[205] W. T. Overman. *Verification of Concurrent Systems: Function and Timing*. PhD thesis, University of California at Los Angeles, 1981.

[206] R. Paige and R. E. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal on Computing* 16(6): 973–989.

[207] D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, pp. 167–183. Springer, 1981.

[208] D. Peled. All from one, one for all: on model checking using representatives. In Courcoubetis [83], pp. 409–423.

[209] D. Peled. Combining partial order reductions with on-the-fly model-checking. In Dill [97], pp. 377–390.

[210] D. Peled. Verification for robust specification. In Elsa Gunter, ed., *Conference on Theorem Proving in Higher Order Logic*, pp. 231–241. Springer, 1997.

[211] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the nexttime operator. *Information Processing Letters*, 1997.

[212] D. Peled, T. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of ω-regular languages. In *CONCUR'96, 7th International Conference on Concurrency Theory*, *LNCS* 1119, pp. 596–610. Springer, 1996.

[213] C. Pixley. Introduction to a computational theory and implementation of sequential hardware equivalence. In Clarke and Kurshan [71], pp. 54–64.

[214] C. Pixley, G. Beihl, and E. Pacas-Skewes. Automatic derivation of FSM specification to implementation encoding. In *Proceedings of the International Conference on Computer Design*, pp. 245–249, Cambridge, MA, October 1991.

[215] C. Pixley, S-W. Jeong, and G. D. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *Proceedings of the 29th Design Automation Conference*, pp. 620–623, June 1992.

[216] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*, pp. 46–57. IEEE Computer Society Press, 1977.

[217] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science* 13: 45–60.

[218] A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, ed., *Logics and Models of Concurrent Systems*, *NATO ASI* 13. Springer, 1984.

[219] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pp. 337–350.

[220] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In Wolper [249], pp. 84–97.

[221] R. Rajkumar. *Task Synchronization in Real-Time Systems*. PhD thesis, Carnegie Mellon University, 1989.

[222] Tomas G. Rokicki and Chris J. Myers. Automatic verification of timed circuits. In Dill [97], pp. 468–480.

[223] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, ed., *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pp. 353–378. Prentice Hall, 1994.

[224] V. Roy and R. de Simone. Auto/Autograph. In Clarke and Kurshan [71], pp. 235–250.

[225] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer Aided Design*, Santa Clara, Ca., November 1993.

[226] S. Safra. On the complexity of omega-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pp. 319–327. IEEE Computer Society, 1988.

[227] R. Schlor and W. Damm. Specification and verification of system-level hardware designs using timing diagrams. In *EDAC 93*, 1993.

[228] L. Sha, M. H. Klein, and J. B. Goodenough. Rate monotonic analysis for real-time systems. In *Foundations of Real-Time Computing—Scheduling and Resource Management*. Kluwer Academic, 1991.

[229] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In Sifakis [231], pp. 157–165.

[230] G. Shurek and O. Grumberg. The modular framework of computer-aided verification: Motivation, solutions and evaluation criteria. In Clarke and Kurshan [71], pp. 214–223.

[231] J. Sifakis, ed. *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems*, *LNCS* 407. Springer, 1989.

[232] A. P. Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, Harvard University, 1983.

[233] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM* 32(3): 733–749.

[234] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science* 49: 217–237.

[235] Richard H. Sloan and Ugo Buy. Stubborn sets for real-time Petri nets. *Formal Methods in System Design* 11(1): 23–40.

[236] C. Stirling and D. J. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science* 89(1): 161–177.

[237] I. Suzuki. Proving properties of a ring of finite-state machines. *IPL* 28: 213–214.

[238] N. Suzuki, ed. *Symbolic Computation Algorithms on Shared Memory Multiprocessors*. MIT Press, 1992.

[239] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing* 1: 146–160.

[240] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics* 5: 285–309.

[241] (TCAS) Minimum operational performance standards for traffic alert and collision avoidance system (TCAS) airborne equipment, volume II. Radio Technical Commission for Aeronautics, September 1990.

[242] J. D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, 1984.

[243] G. von Bochmann and D. K. Probst, eds. *Workshop on Computer-Aided Verification. Fourth International Workshop, CAV'92. Proceedings*, *LNCS* 663. Springer, 1992.

[244] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd Workshop on Computer Aided Verification*, *LNCS* 531, pp. 156–165. Springer 1990.

[245] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In LICS86 [174], pp. 332–344.

[246] P. D. Vigna and C. Ghezzi. Context-free graph grammars. *Information and Computation* 37: 207–233.

[247] G. Winskel. Model checking in the modal ν-calculus. In *Proceedings of the 16th International Colloquium on Automata, Languages, and Programming*, *LNCS* 372, pp. 761–772. Springer, 1989.

[248] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Language*, January 1986.

[249] P. Wolper, ed. *Proceedings of the 1995 Workshop on Computer-Aided Verification*, *LNCS* 939. Springer, 1995.

[250] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In Sifakis [231], pp. 68–80.

[251] J. Yang, A. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1993.

[252] T. Yoneda and B. Schlingloff. Efficient verification of parallel real-time systems. *Formal Methods in System Design* 11(2): 197–215.

[253] T. Yoneda, A. Shibayama, B. Schlingloff, and E. M. Clarke. Efficient verification of parallel real-time systems. In Courcoubetis [83], pp. 321–332.