Avoiding The State Explosion Problem In Temporal Logic Model Checking Algorithms

E. M. Clarke and O. Grümberg
Carnegie Mellon University, Pittsburgh

1. Introduction

Many distributed programs can be viewed at some level of abstraction as communicating finite state machines. The dream of somehow using this observation to automate the verification of such programs can be traced all the way back to the early papers on Petri nets in the 1960's ([13], [18]). The temporal logic *model checking procedure* of Clarke, Emerson, and Sistla ([6], [7], [20]) also attempts to exploit this observation. Their algorithm determines whether the global state transition graph associated with some concurrent program satisfies a formula in the temporal logic *CTL*. The algorithm is linear in both the size of the global state graph and the length of the specification and has been used successfully to find errors in network protocols and asynchronous circuits designs ([4], [9], [17]). A number of other researchers have extended the basic model checking algorithm or proposed alternative algorithms ([1], [3], [12], [15], [19], [22]). Although these algorithms differ significantly in the type of logic that is used and in the way that issues like fairness are handled, they all suffer from one apparently unavoidable problem: In analyzing a system of N processes, the number of states in the global state graph may grow exponentially with N. We call this problem the *state explosion problem.* Our approach to this problem is based on another observation about distributed programs. Although a given program may involve a large number of processes, it is usually possible to partition the processes into a small number of equivalence classes so that all of the processes in a given class are essentially identical. Thus, by devising techniques for automatically reasoning about systems with many identical processes, it may be possible to make significant progress on the general problem.

In [8] we addressed the problem of devising an appropriate logic for reasoning about networks with many identical processes. The logic that we proposed is based on computation trees and is called *Indexed CTL**, or ICTL*. It includes all of CTL* ([7], [10], [11]) with the exception of the nexttime operator and can, therefore, handle both linear and branching time properties with equal facility. Typical operators include EFf, which will hold in a state provided that f eventually holds along some computation path starting from that state and AFf, which will hold in a state provided that f eventually holds along all computation paths. In addition, our logic permits formulas of the form $\bigwedge_i f(i)$ and $\bigvee_i f(i)$ where $f(i)$ is a formula of our logic. All of the atomic propositions that appear within the subformula $f(i)$ must be subscripted by i. A formula of our logic is said to be *closed* if all indexed propositions are within the scope of either a $\bigwedge_i$ or $\bigvee_i$. A *model* for our logic is a labelled state transition graph or *Kripke structure* that represents the possible global state transitions of some network of finite-state processes. For a network of N processes this state graph may be obtained as a product of the state graphs of the individual processes. Instances of the same atomic proposition in different processes are distinguished by using the number of the process as a subscript; thus, A_5 represents the instance of atomic proposition A associated with the fifth process.

Since a closed formula of our logic cannot contain any atomic propositions with constant index values, it is impossible to refer to a specific process by writing such a formula. Hence, changing the number of processes in a family of identical processes should not effect the truth of a formula in our logic. In [8] we showed how to make this intuitive idea precise by introducing a new notion of *bisimulation* [16] between two Kripke structures with the same set of indexed propositions but different sets of index values. We proved that if two structures correspond in this manner, a closed formula of ICTL* will be true in the initial state of one if and only if it is true in the initial state of the other. We say that the two structures are *ICTL***-equivalent.* In order to set up the bisimulation between the two Kripke structures, however,

This research was partially supported by NSF Grant MCS-82-16706. The second author, O. Grümberg, is currently on leave from Technion, Haifa and is partially supported by a Weizmann postdoctoral fellowship.

it is necessary to have an explicit representation of their state transition relations. Thus, while the results in [8] are a necessary first step, they do not completely solve the state explosion problem. Sistla and German [21] have attempted to remedy this problem, but their algorithm runs in triple exponential time and would be quite difficult to implement.

The approach that we use in this paper avoids the problem of having to explicitly construct the bisimulation relation. Suppose that M_k has k identical copies of process P, i.e. $M_k = M_0 \times P^k$. We can think of $M = \{M_1, M_2, \ldots\}$ as a distributed *algorithm* with each M_k representing an instance of the algorithm for a different number of processes. What we would like to do is to compute the first few Kripke structures in the sequence $M_1, M_2, \ldots$ until we reach a point where M_r and M_{r+1} are ICTL*-equivalent and then conclude by induction that for all $k \geq r$, M_k and M_r will be ICTL*-equivalent. Unfortunately, this scheme doesn't quite work. In Section 4 we show that it is possible to select M_0 and P in such a way that M_1 is ICTL*-equivalent to M_2, but M_2 is not ICTL*-equivalent to M_3. It is not enough to show that M_r and M_{r+1} have the same behavior. In addition, we must somehow force $M_r \times P^n$ and $M_{r+1} \times P^n$ to have the same behavior for every $n \geq 0$.

We accomplish this by constructing a single process P^* called the *closure* of P whose states are abstractions of states in P^n. We prove that if $M_r \times P^*$ and $M_{r+1} \times P^*$ are equivalent under a suitable notion of equivalence, then for all $k \geq r$, M_k and M_r will be ICTL*-equivalent. We call this result the *collapsing theorem for networks with many identical processes.* When M has the property that for all $k \geq r$, M_k and M_r are ICTL*-equivalent, then we say that the algorithm M is *r-reducible.* By using the collapsing theorem it is possible to reduce an infinite set of verification problems to a single problem! Thus, we are able to prove that the algorithm M satisfies some ICTL* specification in general by considering only a finite number of instances of the algorithm. Another virtue of our approach is its simplicity. Although some creativity will in general be required to obtain an effective closure for a given algorithm M, the check that $M_r \times P^*$ and $M_{r+1} \times P^*$ are equivalent is easily automatible.

Our paper is organized as follows: Section 2 describes the model for networks of finite state processes that we use in the paper. Section 3 reviews the syntax and semantics of the logic CTL* and its extension ICTL* for reasoning about systems of identical processes. This section also gives the definition of ICTL* equivalence. In Section 4 we state the basic properties that process closures should have and give an appropriate notion of equivalence for structures obtained from such processes. Section 5 contains the *collapsing theorem* discussed above. In Section 6 we give a polynomial algorithm for determining equivalence between process closures and show that the ideas in Section 5 lead to an effective verification procedure. In Section 7 we show how the collapsing theorem can be used to verify two simple concurrent algorithms. The paper concludes in Section 8 with a discussion of some possible extensions of this work.

2. Finite State Processes.

Our model of computation is similar to the CCS model used by Milner [16]. Let A be a set of primitive *open actions* such that $\bar{a} \in A$ whenever $a \in A$ and $\bar{\bar{a}} = a$. The set *ACT* of *process actions* contains the open actions in A, a special action λ used for transitions that do not require syncronization, and *syncronization actions* of the form $a\bar{a}$ where a is in A. The λ action and the synchronization actions are called *completed actions.*

A *process* P is a 5-tuple $P = \langle AP, S, R, s_0, L \rangle$ where,

- AP is the set of atomic propositions.
- S is the set of states.
- $R \subseteq S \times ACT \times S$. We write $s_1 \xrightarrow{\alpha} s_2$ to indicate that $(s_1, \alpha, s_2) \in R$.
- $s_0 \in S$ is the initial state.
- $L : S \rightarrow \mathcal{P}(AP)$ is a function that labels each state with a set of atomic proposition.

A *path* π is a sequence of states $s_1, s_2, \ldots$ such that for each i there exists a completed action α with $s_i \xrightarrow{\alpha} s_{i+1}$.

Let $P_1 = \langle AP_1, S_1, R_1, s_0^1, L_1 \rangle$ and $P_2 = \langle AP_2, S_2, R_2, s_0^2, L_2 \rangle$ be two processes. The *product process* $P_1 \times P_2 = \langle AP, S, R, s_0, L \rangle$ is defined as follows:

- AP is the disjoint union of AP_1 and AP_2.
- $S = S_1 \times S_2$.
- R will contain two types of transitions.

  ∘ $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$ iff $[s_1 \xrightarrow{a} s_1'$ and $s_2 = s_2']$ or $[s_2 \xrightarrow{a} s_2'$ and $s_1 = s_1']$, where a is either an open action or the λ action.

  ∘ $(s_1, s_2) \xrightarrow{a\bar{a}} (s_1', s_2')$ iff $[s_1 \xrightarrow{a} s_1'$ and $s_2 \xrightarrow{\bar{a}} s_2']$ or $[s_1 \xrightarrow{a\bar{a}} s_1'$ and $s_2 = s_2']$ or $[s_2 \xrightarrow{a\bar{a}} s_2'$ and $s_1 = s_1']$.

- $s_0 = (s_0^1, s_0^2)$
- L: $S_1 \times S_2 \rightarrow \mathcal{P}(AP)$ such that $L((s_1, s_2))$ is the disjoint union of $L_1(s_1)$ and $L_2(s_2)$.

We define the product P^n to be $(\ldots(P_1 \times P_2) \times \ldots) \times P_{n-1}) \times P_n)$ where each P_i is a copy of P with the atomic propositions that label the states indexed by i. The action names are uneffected by this indexing. In this case we say that P^n is a *process with index set* $I = \{1, \ldots, n\}$. A state σ in P^n can either be viewed as an n-tuple $(s_1, \ldots, s_n)$ or as a pair $((s_1, \ldots, s_{n-1}), s_n)$ where s_i is the component of process i. We will also use the convention that $\sigma|_i$ is s_i, the i^{th} component of the n-tuple representation of σ.

Let M and P be as shown in Figure 2-1, then the product $M \times P$ is shown in Figure 2-2.

Intuitively, a distributed *algorithm* consists of a finite set of component processes with some rules for connecting these processes together to form networks of different sizes. In this paper we consider a simple but important class of distributed algorithms such that in each instance of the algorithm all but a finite number of the processes are identical and each process can communicate with every other process. We represent an instance of such an algorithm by a product of the form $M_r = M_0 \times P^r$ for $r > 0$, where M_0 gives the combined behavior of the component processes that are not identical. We expect that our results also hold for distributed algorithms with more complicated rules for combining component processes.

3. Indexed CTL*

There are two types of formulas in CTL*: state formulas (which are true in a specific state) and path formulas (which are true along a specific path). Since our logic is based on CTL*, we will have the same two types of formulas. Let AP be a set of proposition names, which are indexed by a finite set of indices I (a subset of $\mathbf{N}$). A state formula is either:

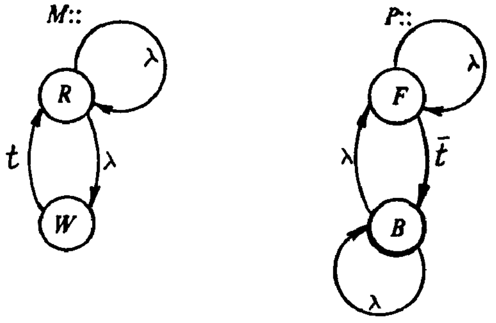

Figure 2-1: Two Finite State Processes: M and P.

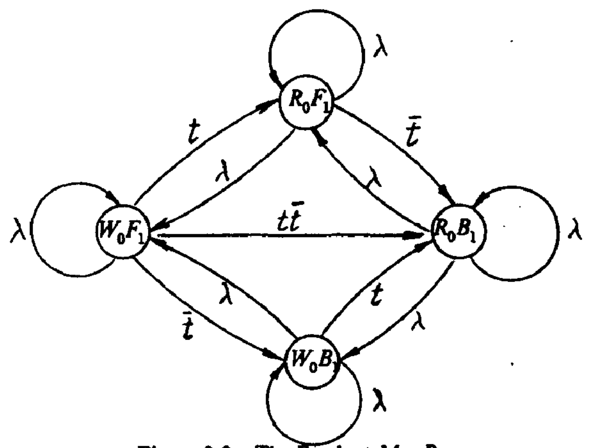

Figure 2-2: The Product $M \times P$.

- A_i, if $A \in AP$ and $i \in I$.
- If f and g are state formulas, then $\neg f$ and $f \vee g$ are state formulas. Moreover, if f has exactly one free index variable i, then $\bigvee_i f$ is a state formula. (We will write $f(i)$ to indicate that f has a free index variable i.)
- If f is a path formula, then $\mathbf{E}(f)$ is a state formula.

A path formula is either:

- A state formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $\bigvee_i f(i)$, and $f \mathbf{U} g$ are path formulas.

We define the semantics of Indexed CTL* with respect to a structure $K = \langle AP, I, S, R, s_0, L \rangle$, where

- AP is the set of atomic formulas.
- I is the set of index values (a subset of $\mathbf{N}$).
- S is a set of states.
- $R \subseteq S \times S$ is the transition relation, which must be total in both of its arguments. We write $s_1 \to s_2$ to indicate that $(s_1, s_2) \in R$.
- s_0 is the initial state.
- L: $S \to \mathcal{P}(AP \times I)$ is the proposition labeling. We will write A_i instead of (A, i).

$L_i(s)$ will be the restricton of L to the set of atomic formulas indexed by i. We only consider transition relations where every state is reachable from the initial state. We define a *path in K* to be a sequence of states, $\pi = s_1, s_2, \ldots$ such that for every $i \geq 1$, $s_i \to s_{i+1}$. π^i will denote the suffix of π starting at s_i.

Note that structures are different from processes. A structure may be obtained from a process with index set I by restricting the transition relation of the process so that only transitions on completed actions are allowed. Also, if some state in the process has no transitions on completed actions, we add to the corresponding state in the structure a transition from that state back to itself. It will sometimes be convenient to refer to a process in a context which requires a structure instead. When this happens, the required structure is the one obtained from the process by the above conventions.

We use the standard notation to indicate that a state formula f holds in a structure: $K, s \models f$ means that f holds at state s in structure K. Similarly, if f is a path formula, $K, \pi \models f$ means that f holds along path π in structure K. The relation $\models$ is defined inductively as follows (assuming that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

1. $s \models A_i \quad \Leftrightarrow \quad A_i \in L(s)$.
2. $s \models \neg f_1 \quad \Leftrightarrow \quad s \not\models f_1$.
3. $s \models f_1 \vee f_2 \quad \Leftrightarrow \quad s \models f_1$ or $s \models f_2$.

4. $s \models \bigvee_i f_1(i)$ $\Leftrightarrow$ there exists an $i_0 \in I$ such that $s \models f_1(i_0)$.

5. $s \models E(g_1)$ $\Leftrightarrow$ there exists a path π starting with s such that $\pi \models g_1$.

6. $\pi \models f_1$ $\Leftrightarrow$ s is the first state of π and $s \models f_1$.

7. $\pi \models \neg g_1$ $\Leftrightarrow$ $\pi \not\models g_1$.

8. $\pi \models g_1 \vee g_2$ $\Leftrightarrow$ $\pi \models g_1$ or $\pi \models g_2$.

9. $\pi \models \bigvee_i g_1(i)$ $\Leftrightarrow$ there exists an $i_0 \in I$ such that $\pi \models g_1(i_0)$.

10. $\pi \models g_1 U g_2$ $\Leftrightarrow$ there exists a $k \geq 0$ such that $\pi^k \models g_2$ and for all $0 \leq j < k$, $\pi^j \models g_1$.

We have omitted the nexttime operator, since it can be used to count the number of processes. For example, consider a ring of processes that pass around a token. Using the nexttime operator **X**,

$$\bigwedge_i A(t_i \Rightarrow (XXXt_i))$$

says that any process that has the token will receive it again in exactly three steps. This is only true if the ring has exactly three processes.

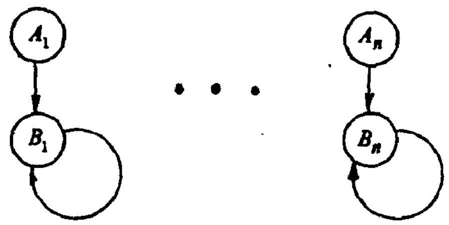

Figure 3-1: Example to Illustrate Restrictions on ICTL*

Even with this restriction on the nexttime operator, the logic is too powerful; by nesting the operators $\bigwedge_i$ and $\bigvee_i$ it might still be possible to count the number of processes in a concurrent system. Suppose we take as our Kripke structure the global state graph for the concurrent program in figure 3-1. The following formula sets a lower bound on the number of processes:

$$\bigvee_i (A_i \wedge EF(B_i \wedge \bigvee_j (A_j \wedge EF(B_j \wedge \bigvee_k (A_k \ldots)))))$$

Once B_i becomes true, it remains true. Therefore, if $\bigvee_k A_k$ is true, we know that this k is different from all of the preceding indices mentioned in the formula. For this reason, we will use a restricted form of ICTL*. The additional restrictions are:

- $\bigvee_i f$ is a permissible state formula only if f does not contain any $\bigvee_j$ operators.
- $\bigvee_i g$ is not a permissible path formula.
- g_1 U g_2 is a permissible path formula only if neither g_1 nor g_2 contains any $\bigvee_j$ operators.

In practice, many of the most interesting properties of networks of identical processes can be expressed in the restricted logic. In the remainder of the paper, we will refer to the restricted logic as ICTL* unless otherwise stated.

We want to be able to define a correspondence (or bisimulation) between two structures, K_1 and K_2 such that if the structures correspond, then one structure satisfies an ICTL* formula if and only if the other satisfies it as well. Since the restrictions to ICTL* do not permit the use of two different indices within an until operator, it is impossible to refer to the behavior of two different processes along a specific path. Thus, the notion of correspondence between structures only needs to refer to one index from each structure at a time. Because of this, we define a set of finite correspondence relations, $C_{ii'} \subseteq S_1 \times S_2$, that relate the behavior of an index i in I_1 to the behavior of an index i' in I_2. Intuitively, (s,s') is in $C_{ii'}$ if index i in the state s behaves like index i' in the state s'.

We may have portion of a path along which the behavior of i does not change, i.e. several consecutive states are all labelled by the same set of propositions indexed by i. This type of behavior is called *stuttering* ([5], [8], [14]). We will call such a sequence of states an *i-block*. Since ICTL* has no nexttime operator, it is impossible to differentiate between a single state and an i-block with the same labelling as the state. However, when we correspond a state with an i-block, we must ensure that the i-block is finite. This is similar to the notion of *stuttering equivalence* considered in [5].

$C_{ii'} = \bigcap_n C^n_{ii'}$, where $C^n_{ii'}$ is inductively define as follows:

- $C^0_{ii'} = \{(s_1,s_2) \mid L_i(s_1) = L_{i'}(s_2)\}$
- $s_1 \, C^{n+1}_{ii'} \, s_2$ iff
  - For every path π, starting in s_1, there exists a path π', starting in s_2 and partitions of both paths $B_1, B_2, \ldots$, $B'_1, B'_2, \ldots$ such that for every j, B_j and B'_j are both finite and nonempty. Moreover, $B_j \; C^n_{ii'} \; B'_j$.
  - For every path π', starting in s_2, there exists a path π, starting in s_1 that satisfies the same conditions as above.

Let K_1 and K_2 be two structures with initial states s^1_0 and s^2_0 and index sets I_1 and I_2. Then $K_1 \, C_{ii'} \, K_2$ iff $s^1_0 \, C_{ii'} \, s^2_0$. Moreover, $K_1 \, C \, K_2$ iff there exists an *index relation* $IN \subseteq I_1 \times I_2$, total in both arguments, such that for every $(i,i') \in IN$, $K_1 \, C_{ii'} \, K_2$. The following theorem is proved in [8].

Theorem 1: If $K_1 \, C \, K_2$ then $K_1, s^1_0 \models h \Leftrightarrow K_2, s^2_0 \models h$ for every closed ICTL* formula h.

4. Process Closures

In order to show that some distributed algorithm M is r- reducible, we must find an r such that for every $k > r$, M_k is ICTL*-equivalent to M_r. Unfortunately, it is not sufficient to show that M_r is ICTL*-equivalent to M_{r+1}. If M_0 and P are as shown in Figure 4-1, then M_1 is ICTL*-equivalent to M_2, but M_2 is not ICTL*-equivalent to

M_3. It is not enough to show that M_r and M_{r+1} have the same behavior. In addition, we must require that $M_r \times P^k$ and $M_{r+1} \times P^k$ have the same behavior for every k. We can accomplish essentially the same thing by showing that $M_r \times P^*$ and $M_{r+1} \times P^*$ are equivalent, where P^* is a special process called *the closure of P*. The closure serves as an abstraction for P^k for all $k > 0$ and must be supplied by the person who is doing the verification. We will use M_r^P to denote $M_r \times P^*$. Note that each state σ of S_r^P is a pair (s, p^*) in which the first component s is a state of M_r and the second component p^* is a state of P^*.

The user must also supply two families of homomorphisms $h_k: M_k \rightarrow M_r^P$ for $k \geq r$ and $g_k: M_k \rightarrow M_{r+1}^P$ for $k \geq r+1$. The homomorphisms associate with every computation of M_k a uniquely determined computation of M_r^P (or M_{r+1}^P). The homomorphism h_k will have the following properties:

- it must map the initial state of M_k to the initial state of M_r^P.
- it is the identity on the components 0 through r of the states, i.e. $\sigma|_i = h(\sigma)|_i$ for $i \leq r$.
- If σ_1 is a reachable state of M_k and $\sigma_1 \xrightarrow{\alpha} \sigma_2$ is a transition involving a completed action α in M_k, then there is a transition $h_k(\sigma_1) \xrightarrow{\alpha} h_k(\sigma_2)$ in M_r^P. Furthermore, if α is the syncronization action $a\bar{a}$ and a is taken by the i^{th} process in $\sigma_1 \xrightarrow{\alpha} \sigma_2$ with $i \leq r$, then the i^{th} process will also take a action in $h_k(\sigma_1) \xrightarrow{\alpha} h_k(\sigma_2)$. Otherwise, if $i > r$, then the a action in $h_k(\sigma_1) \xrightarrow{\alpha} h_k(\sigma_2)$ is taken by P^*. A similar restriction also applies to $\bar{a}$ and λ.

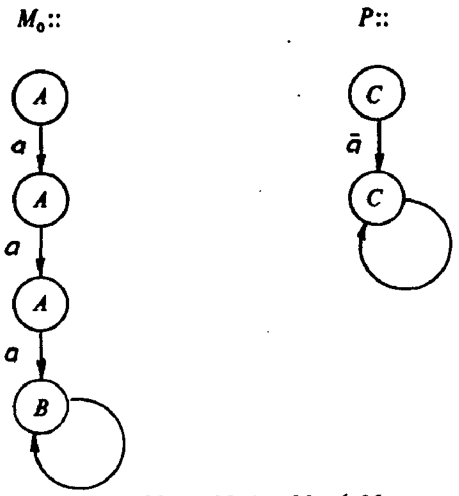

Figure 4-1: $M_1 \equiv M_2$, but $M_2 \not\equiv M_3$.

We wish to define an equivalence relation D between M_r^P and M_{r+1}^P which will ensure that for every k, $M_k \, C \, M_{k+1}$, where C is the relation defined in Section 3. In other words, we must ensure that there is an index relation $IN \subseteq I_k \times I_{k+1}$ such that $M_k \, C_{ii'} \, M_{k+1}$ for every $(i,i') \in IN$. The definition of the equivalence relation D is somewhat more complicated than the one given in Section 3 because of the P^* component. $M_r^P \, D \, M_{r+1}^P$ iff there exists an index relation $IN_1 \subseteq I_r \times I_{r+1}$ such that for every $(i,i') \in IN_1$, $M_r^P \, D_{ii'} \, M_{r+1}^P$ and in addition $M_r^P \, E \, M_{r+1}^P$. The relation $D_{ii'}$ is used in constructing $C_{ii'}$ for $(i,i') \in IN_1$, while the relation E is used in constructing $C_{ii'}$ for $(i,i') \in IN - IN_1$. $D_{ii'}$ and E are defined over $S_r^P \times S_{r+1}^P$. As before, we say that two structures are $D_{ii'}$ or E related to each other if their initial states are. Note again that $\sigma = (s, p^*)$.

$D_{ii'} = \bigcap_n D_{ii'}^n$, where $D_{ii'}^n$ is:

- $D_{ii'}^0 = \{(\sigma_1, \sigma_2) \mid L_i(s_1) = L_{i'}(s_2) \wedge p_1^* = p_2^*\}$
- $\sigma_1 \, D_{ii'}^{n+1} \, \sigma_2$ iff
    - For every path π, starting in σ_1, there exists a path π', starting in σ_2 and partitions of both paths $B_1, B_2, \ldots, B_1', B_2', \ldots$ such that for every j:
        1. B_j, B_j' are nonempty, finite, and defined along actions in M_r and M_{r+1}, respectively.
        2. $B_j \, D_{ii'}^n \, B_j'$
        3. Let t_j be the transition $last(B_j) \rightarrow first(B_{j+1})$. Then either t_j is a transition in M_r and t_j' is a transition in M_{r+1} or, if t_j involves some action in P^*, then t_j' involves exactly the same action in P^*.
    - For every path π', starting in σ_2, there exists a path π, starting in σ_1 that satisfies the same conditions as above.

$E = \bigcap_n E^n$, where E^n is defined exactly like $D_{ii'}^n$ except that the basis case is given by $E^0 = \{(\sigma_1, \sigma_2) \mid p_1^* = p_2^*\}$.

5. The Collapsing Theorem

We now state the collapsing theorem for r- reducible algorithms.

Theorem 2: If $M_r^P \, D \, M_{r+1}^P$ then for every $k \geq r$, $M_k \, C \, M_r$.

Proof: We prove that for every $k \geq r$, $M_k \, C \, M_{k+1}$. For each k we must show that there exists a relation $IN \subseteq I_k \times I_{k+1}$ such that $(i, i') \in IN$ implies that $M_k \, C_{ii'} \, M_{k+1}$. We consider first the case in which $k = r$. We already know that $(M_r \times P^*) \, D \, (M_{r+1} \times P^*)$. This means that there exists a relation $IN_1 \subseteq I_r \times I_{r+1}$ such that for $(i, i') \in IN_1$, $(M_r \times P^*) \, D_{ii'} \, (M_{r+1} \times P^*)$. Let p_0^* be the start state of P^*. We prove the following statement by induction on n:

> Let $s \in M_r$, $s' \in M_{r+1}$, $\bar{s} = (s, p_0^*) \in M_r \times P^*$, and $\bar{s}' = (s', p_0^*) \in M_{r+1} \times P^*$, with $\bar{s} \, D_{ii'}^n \, \bar{s}'$, then $s \, C_{ii'}^n \, s'$.

The basis case is easy to prove: $\bar{s} \, D_{ii'}^0 \, \bar{s}'$ implies that $L_i(s) = L_{i'}(s')$. This in turn implies that $s \, C_{ii'}^0 \, s'$. Next, assume that $\bar{s} \, D_{ii'}^{n+1} \bar{s}'$ holds. We must prove that $s \, C_{ii'}^{n+1} s'$ also holds.

Thus, let π be a path that starts at s in M_r. Let $\bar\pi$ be the sequence obtained from π by replacing each state s_m on π by (s_m, p_0^*). All transitions along $\bar\pi$ are M_r transitions. $\bar\pi$ is a path starting at $\bar s$, so there exists a path $\bar\pi'$ starting at $\bar s'$ and partitions of both paths $\bar B_1, \bar B_2, \ldots, \bar B_1', \bar B_2', \ldots$ such that the conditions in the definition of $D_{ii'}^{n+1}$ hold. Because of the third condition we know that each state of $\bar\pi'$ has the form (s_m', p_0^*) and all transitions along $\bar\pi'$ are M_{r+1} transitions.

Let π' be the path obtained by deleting the p_0^* component of each state in $\bar\pi'$. Let $B_1, B_2 \ldots, B_1', B_2', \ldots$ be the partitions of π and π' determined by the partitions of $\bar\pi$ and $\bar\pi'$. It is easy to see that π and π' satisfy the conditions in the definition of the $C_{ii'}^{n+1}$ relation. For example, $B_j \, C_{ii'}^n \, B_j'$ follows immediately from the inductive hypothesis and the fact that $\bar B_j \, D_{ii'}^n \, \bar B_j'$. Essentially the same argument can be used to obtain a path π starting at s, given a path π' starting at s'. We see that $M_r \, C_{ii'} \, M_{r+1}$ by applying the inductive hypothesis to the case in which s is the start state of M_r and s' is the start state of M_{r+1}.

Next, we show that $(M_r \times P^*) \, D \, (M_{r+1} \times P^*)$ implies $M_k \, C \, M_{k+1}$ for $k > r$. We must show that there exists a relation $IN \subseteq I_k \times I_{k+1}$ such that $(i, i') \in IN$ implies $M_k \, C_{ii'} \, M_{k+1}$. We already know that there exists a relation $IN_1 \subseteq I_r \times I_{r+1}$ such that $(M_r \times P^*) \, D_{ii'} \, (M_{r+1} \times P^*)$ for $(i, i') \in IN_1$. Choose $IN = IN_1 \cup \{(r+1, r+2), \ldots, (k, k+1)\}$. There are two major cases:

Case 1: $(i, i') \in \{(r+1, r+2), \ldots, (k, k+1)\}$. Note that with each state $(s_0, s_1, \ldots, s_r, s_{r+1}, \ldots, s_k)$ of M_k we can associate a uniquely determined state $(s_0, s_1, \ldots, s_r, p^*)$ of $M_r \times P^*$ obtained by applying the homomorphism h_k to the first state. Likewise, g_{k+1} applied to a state in M_{k+1} gives the analogous state in $M_{r+1} \times P^*$. The homomorphisms extend in the obvious way to sequences of states in M_k or in M_{k+1}.

The following inductive hypothesis will enable us to construct a path in M_{k+1} given a path in M_k, using h_k:

> Let $\bar s$ be a state of $M_r \times P^*$ and $\bar s'$ be a state of $M_{r+1} \times P^*$ with $\bar s \, E^n \, \bar s'$. Let s be a state of M_k such that $h_k(s) = \bar s$, and let s' be a state that agrees with s on its last $k-r$ components and agrees with $\bar s'$ on components 0 through $r+1$. Then $s \, C_{ii'}^n \, s'$.

Figure 5-1 illustrates the notational conventions. A similar argument will enable us to construct a path in M_k given a path in M_{k+1}, using g_{k+1}. Note that if $\bar s \, E^n \, \bar s'$ then their P^*-components are equal. The basis case of the inductive hypothesis follows immediately from the requirement that s and s' agree on their last $k-r$ components. For the induction step we show that if $\bar s, \bar s', s, s'$ are as above with $\bar s \, E^{n+1} \, \bar s'$ then $s \, C_{ii'}^{n+1} s'$. Let π be a path starting at s and let

$$\begin{tikzcd}
M_r \times P^*: \ \bar s = (s_0, \ldots, s_r, p^*) \arrow[d, phantom, "|||_E"] & M_k: \ s = (s_0, \ldots, s_r, \underbrace{s_{r+1}, \ldots, s_k}) \arrow[l, "h_k"'] \arrow[d, dashed] \\
M_{r+1} \times P^*: \ \bar s' = (\underbrace{s_0', \ldots, s_{r+1}'}, p^*) \arrow[r, dashed] & M_{k+1}: \ s' = (\underbrace{s_0', \ldots, s_{r+1}'}, \overbrace{s_{r+1}, \ldots, s_k})
\end{tikzcd}$$

Figure 5-1: The Construction of π' starting at s', given π starting at s.

$\bar\pi = h_k(\pi)$. Since h_k is a homomorphism $\bar\pi$ is a path in $M_r \times P^*$. $\bar\pi$ starts at $\bar s$ so there is a path $\bar\pi'$ starting at $\bar s'$ that satisfies the definition of E^{n+1} including the existence of partitions $\bar B_1, \bar B_2, \ldots, \bar B_1', \bar B_2', \ldots$ of $\bar\pi$ and $\bar\pi'$. The partition $\bar B_1, \bar B_2, \ldots$ of $\bar\pi$ determines a partition $B_1, B_2, \ldots$ of π in the obvious way. The mth state of π is the first (last) state of B_j if and only if the mth state of $\bar\pi$ is the first (last) state of $\bar B_j$.

For the construction of π' it is necessary to know that for each j all of the states in block B_j agree on the last $k-r$ components. To see that this is true, let s_m and s_{m+1} be two adjacent states on π that do not agree on the last $k-r$ components. This can happen only if the transition from $\bar s_m = h(s_m)$ to $\bar s_{m+1} = h(s_{m+1})$ in $\bar\pi$ involves P^*. Transitions involving P^* can only occur at block boundaries by the first condition in the definition of $D_{ii'}$. It follows that s_m and s_{m+1} can't both be in the same block B_j of π.

We construct a sequence π' starting at s' as follows: Let s_m' be the m th state on π'. s_m' will agree with $\bar s_m'$ on components 0 through $r+1$. If $\bar s_m'$ is in block $\bar B_j'$, then s_m' will agree with the states of B_j in its last $k-r$ components (see Figure 5-1). We must show that the sequence π' is really a path in M_{k+1}. Let s_m' and s_{m+1}' be two consecutive states on π'. We show that there is a transition in M_{k+1} from s_m' to s_{m+1}'. Let $\bar s_m'$ and $\bar s_{m+1}'$ be the corresponding states on $\bar\pi'$. If $\bar s_m'$ and $\bar s_{m+1}'$ are both in the same block $\bar B_j'$ then s_m' and s_{m+1}' will agree on the last $k-r$ components. Since the transition from $\bar s_m'$ to $\bar s_{m+1}'$ within $M_{r+1} \times P^*$ only involves processes 0 through $r+1$, there must be a transition from s_m' to s_{m+1}' in M_{k+1}.

Next, assume that $\bar s_m'$ is the last state of block $\bar B_j'$ and that $\bar s_{m+1}'$ is the first state in block $\bar B_{j+1}'$. Let $\bar s_n$ (s_n) be the last state of block $\bar B_j$ (block B_j), and let $\bar s_{n+1}$ (s_{n+1}) be the first state of block $\bar B_{j+1}$ (block B_{j+1}). There are a number of subcases this time.

1. The transition from $\bar{s}'_m$ to $\bar{s}'_{m+1}$ only involves processes 0 through $r+1$. P^* is not involved. In this case the transition from $\bar{s}_n$ to $\bar{s}_{n+1}$ only involves processes 0 through r. Hence, s_n and s_{n+1} must agree on their last $k-r$ components. It follows that s_m and s_{m+1} must also agree on their last $k-r$ components. Thus, this case reduces to the one that we have just considered.

2. The transition from $\bar{s}'_m$ to $\bar{s}'_{m+1}$ only involves the P^* component. Thus, s'_m and s'_{m+1} agree on components 0 through $r+1$. It follows that $\bar{s}_n$ and $\bar{s}_{n+1}$ agree on components 0 through r and that the transition from s_n to s_{n+1} only involves the last $k-r$ processes. Since the last $k-r$ components of s'_m (s'_{m+1}) are identical to the last $k-r$ components of s_n (s_{n+1}) there must be a transition from s'_m to s'_{m+1}.

3. The transition from $\bar{s}'_m$ to $\bar{s}'_{m+1}$ involves P^* and some process numbered 0 to $r+1$. $\bar{s}_n$ and $\bar{s}'_m$ ($\bar{s}_{n+1}$ and $\bar{s}'_{m+1}$) have the same P^* component. The transition from s_n to s_{n+1} corresponds to a joint transition by two processes i and j with $i \le r$ and $j > r$. Without loss of generality assume that process i makes the transition $u_1 \xrightarrow{c} u_2$ and that process j makes transition $v_1 \xrightarrow{\bar{c}} v_2$. If p^*_n (p^*_{n+1}) is the P^* component of $\bar{s}_n$ (of $\bar{s}_{n+1}$), then by the third property in the definition of a homomorphism there must be a $\bar{c}$ transition from p^*_n to p^*_{n+1}. It follows that there must be some process $i' < r+1$ that is enabled to make a c transition from u'_1 to u'_2 in $\bar{s}'_m$ and that the state of this process is u'_2 in $\bar{s}'_{m+1}$. Thus, s'_{m+1} follows from s'_m by a joint $c\bar{c}$ transition in which process $j+1$ moves from state v_1 to state v_2 and process i' moves from u'_1 to u'_2.

It is easy to see that π' satisfies the requirements in the definition of a correspondence relation. $B_j\ C^n_{ii'}\ B'_j$ follows immediately from the inductive hypothesis and the fact that $\bar{B}_j\ E^n\ \bar{B}'_j$. The same argument can be used to obtain a path π starting at s given a path π' starting at s'. This shows that $s\ C^{n+1}_{ii'} s'$ as required to establish the inductive hypothesis. Note that by the inductive hypothesis with $\bar{s}$ the initial state of $M_r \times P^*$, $\bar{s}'$ the initial state of $M_{r+1} \times P^*$, s the initial state of M_k, and s' the initial state of M_{k+1}, we get that $s\,C_{ii'}\,s'$. It follows that $M_k\,C_{ii'}\,M_{k+1}$.

Case 2: We must show that $M_k\,C_{ii'}\,M_{k+1}$ for $(i,i') \in IN_1$. The inductive hypothesis is similar to the one used in the first case:

> Let $\bar{s}$ be a state of $M_r \times P^*$ and $\bar{s}'$ be a state of $M_{r+1} \times P^*$ with $\bar{s}\ D^n_{ii'}\ \bar{s}'$. Let s be a state of M_k such that $h_k(s) = \bar{s}$, and let s' be a state that agrees with s on its last $k-r$ components and agrees with $\bar{s}'$ on components 0 through $r+1$. Then $s\,C^n_{ii'}\ s'$.

Since $\bar{s}\ D^n_{ii'}\ \bar{s}'$, it follows that $\bar{s}\ D^0_{ii'}\ \bar{s}'$. As a consequence, we have that $L_i(\bar{s}) = L_{i'}(\bar{s}')$. The basis case of the inductive hypothesis follows immediately from this observation. For the induction step we show that if $\bar{s}$, $\bar{s}'$, s, s' are as above with $\bar{s}\ D^{n+1}_{ii'}\bar{s}'$, then $s\ C^{n+1}_{ii'} s'$. The proof follows exactly the same lines as in case 1. $M_k\,C_{ii'}\,M_{k+1}$ follows from the inductive hypothesis with $\bar{s}$ the initial state of $M_r \times P^*$, $\bar{s}'$ the initial state of $M_{r+1} \times P^*$, s the initial state of M_k, and s' the initial state of M_{k+1}.

6. Algorithm For Equivalence Between Process Closures

In this section we show how to compute the equivalence relation for closures. We will describe the algorithm for $D_{ii'}$ for $(i,i') \in IN_1$. The algorithm for E will follow the same lines, except that the base case will be constructed according to E^0.

We construct the relation $\mathfrak{D}_{ii'}$ on $S^P_r \times S^P_{r+1}$ that is identical to the relation $D_{ii'}$ defined in Section 4. $\mathfrak{D}_{ii'} = \bigcap_n \mathfrak{D}^n_{ii'}$, where $\mathfrak{D}^n_{ii'}$ is defined as follows:

$$\mathfrak{D}^0_{ii'} = \{ (\sigma_1,\sigma_2) \mid L_i(s_1) = L_{i'}(s_2) \wedge p^*_1 = p^*_2 \}$$

In order to define $\mathfrak{D}^{n+1}_{ii'}$ we must first define the sets of *extended succesors of* σ. We define these sets in terms of the set $ST_{n+1}(\sigma)$ of *the stuttering states of* σ. $ST_{n+1}(\sigma_1)$ is the set of states that are $\mathfrak{D}^n_{ii'}$ related to σ_1 and are reachable from σ_1 along path in which all the transitions are in M_r and all the states are $\mathfrak{D}^n_{ii'}$ related to σ_1. $ST_{n+1}(\sigma_2)$ is defined similarily for transitions within M_{r+1}. We can now define the sets of the extended successors of σ_1 in M^P_r.

- For every action α in P^*, $\alpha - NEXT_{n+1}(\sigma_1)$ is the set of extended successors of σ_1 which are reachable from states in $ST_{n+1}(\sigma_1)$ along a transition that involves an action α in P^*.
- $M - NEXT_{n+1}(\sigma_1)$ is the set of extended successors of σ_1, not included in $ST_{n+1}(\sigma_1)$, which are reachable from states in $ST_{n+1}(\sigma_1)$ along a transition that involves a completed action in M_r.

The sets of the extended successors of σ_2 in M^P_{r+1} are defined similarily. We will also use a predicate $LOOP_{n+1}(\sigma)$ that is true iff there is a cycle containing only states in $ST_{n+1}(\sigma)$.

Let x denote either an action α of P^*, or M. Then we can define $\mathfrak{D}^{n+1}_{ii'}$ as follows:

$$\begin{aligned}\mathfrak{D}^{n+1}_{ii'} = \{(\sigma_1,\sigma_2) \mid\ & LOOP_{n+1}(\sigma_1) = LOOP_{n+1}(\sigma_2) \ \wedge\ \sigma_1\,\mathfrak{D}^n_{ii'}\ \sigma_2 \ \wedge \\ & \bigwedge_x \forall \sigma'_1 \in x-NEXT_{n+1}(\sigma_1)\ \exists \sigma'_2 \in x-NEXT_{n+1}(\sigma_2)\,[\,\sigma'_1\,\mathfrak{D}^n_{ii'}\ \sigma'_2\,] \ \wedge \\ & \bigwedge_x \forall \sigma'_2 \in x-NEXT_{n+1}(\sigma_2)\ \exists \sigma'_1 \in x-NEXT_{n+1}(\sigma_1)\,[\,\sigma'_1\,\mathfrak{D}^n_{ii'}\ \sigma'_2\,]\}\end{aligned}$$

In the journal version of this paper we will show how a low-order polynomial algorithm can be extracted from this construction.

7. Examples

To illustrate how the algorithm in Section 6 might be used we consider two very simple examples. The first consists of a *master* process M_0 and several *slaves* P_i as shown in Figure 2-1. The master process will determine that a job needs to be performed and then start the job on a slave that is not busy. Thus, the master will remain in its *ready* state (R_0) until a job needs to be performed. It will then make a transition to its *waiting* state W_0 and try to rendezvous with a slave (P_i) that is in its *free* state (F_i). The joint transition will cause the master to return to its ready state and the slave to enter its *busy* state (B_i). When the slave has completed the job it will return to its free state.

We will show that the algorithm $M = \{ M_0, M_1, \ldots \}$ with $M_k = M_0 \times P^k$ for $k \geq 1$ is *1-reducible*, i.e. that $M_k \, C \, M_1$ for all $k \geq 1$. In order to demonstrate that this is true we must find a suitable closure P^* together with two sets of homomorphisms $h_k : M_k \rightarrow M_1^P$ for $k \geq 1$ and $g_k : M_k \rightarrow M_2^P$ for $k \geq 2$ that satisfy the conditions given in Section 4.

Intuitively, the states of P^* are abstractions of the states of P^k that are reachable when P^k is run in parallel with M_r. In this case we choose the states of P^* to be sets of states of P. The state $\{ F \}$ of P^* represents a state of P^k in which all k processes are in state F. The state $\{ B \}$ represents a state of P^k in which all of the processes are in the state B. The third and last state handles the case in which some processes of P^k are in state B and some are in state F. The transition graph for process P^* is shown in Figure 7-1. Note that there is a transition from one state to another in P^* iff the same transition occurs between corresponding states of P^k for some $k > 0$.

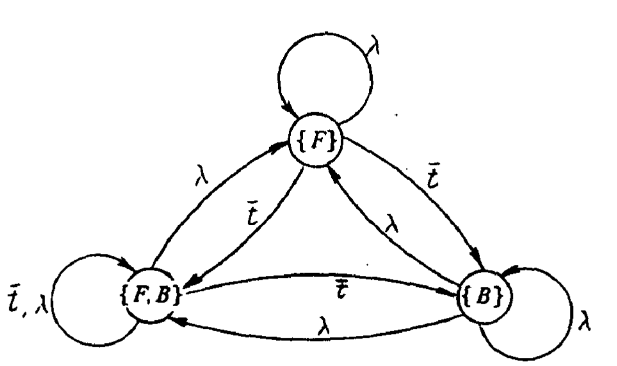

Figure 7-1: The Closure of P for the Master-Slave Algorithm.

The homomorphism h_k is also based on the intuition in the previous paragragh and is given by $h_k(s_0, \ldots, s_r, s_{r+1}, \ldots, s_k) = (s_0, \ldots, s_r, \{ s_{r+1}, \ldots, s_k \})$. Essentially the same definition can be used for g_k with $r+1$ replacing r and $k+1$ replacing k. It is easy to see that h_k and g_k satisfy the first two conditions in the definition of a homomorphism. It is not difficult to establish the third condition as well since any open or completed action that can be made by one of the last k-r processes in some state of M_k is also possible in the P^*-component of the corresponding state of $M_r \times P^*$.

The algorithm in Section 6 can be used to show that $(M_1 \times P^*) \, D \, (M_2 \times P^*)$. Since $M_1 \times P^*$ has 12 states and $M_2 \times P^*$ has 24 states, the computation is tedious but straightforward. By Theorem 2 it follows that M_k and M_1 satisfy the same ICTL* formulas for all $k \geq 1$. In order to determine if some particular formula holds for M_k with $k \geq 1$, the temporal logic model checking procedure described in [7] can be used to check the formula for M_1.

The construction that we used to obtain the closure of the slave process in the example can be generalized. Let P be a process. The *closure of P*, P^*, is defined by: $P^* = \langle AP, S^*, R^*, s_0^*, L^* \rangle$ where $S^* = \mathcal{P}(S) - \{\emptyset\}$. Intuitively, the states of P^* are abstractions of states of P^n. The state $\{s_1, \ldots, s_k\}$ indicates that at least one process of P^n is in each s_i and that each of the processes is in one of the s_i. There are several cases in the definition of R^*. Let $q = \{s_1, \ldots, s_k\} \in S^*$. For every transition $s_i \xrightarrow{a} s_i'$, R^* will include two transitions of the form $q \xrightarrow{a} q'$. The first transition in which $q' = (q - \{s_i\}) \cup \{s_i'\}$ assumes that there is exactly one process in the state s_i. The second transition in which $q' = q \cup \{s_i'\}$ assumes that there are several processes in state s_i.

If two transitions $s_i \xrightarrow{a} s_i'$ and $s_j \xrightarrow{\bar{a}} s_j'$ are possible in state q for $i = j$, then there will be two transitions of the form $q \xrightarrow{a\bar{a}} q'$. The first with $q' = (q - \{s_i\}) \cup \{s_i', s_j'\}$ represents the case in which exactly two processes are in s_i. The second with $q' = q \cup \{s_i', s_j'\}$ represents the case in which more than two processes are in s_i.

If two transitions $s_i \xrightarrow{a} s_i'$ and $s_j \xrightarrow{\bar{a}} s_j'$ are possible in state q for $i \neq j$, then there will be four transitions of the form $q \xrightarrow{a\bar{a}} q'$. The first with $q' = (q - \{s_i, s_j\}) \cup \{s_i', s_j'\}$ represents the case in which exactly one process is in s_i and exactly one process is in s_j. The second with $q' = q \cup \{s_i', s_j'\}$ represents the case in which several processes are in s_i and also in s_j. The two remaining cases with $q' = (q - \{s_i\}) \cup \{s_i', s_j'\}$ and $q' = (q - \{s_j\}) \cup \{s_i', s_j'\}$ represent cases in which exactly one process is in one of the two states but several are in the other. The initial state of P^* is $s_0^* = \{s_0\}$. The labelling function for propositions is given by $L^*(q) = \bigcup_{i=1}^{k} L(s_i)$. The size of P^* is at worst exponential in the size of P.

There are two obvious problems with this definition for the closure of P. The closure of P may be quite large, even if P is very small. Secondly, P^* may contain states that are not reachable in any computation of P^k and behave differently when composed with M_r and M_{r+1}. These problems may be avoided in many cases by considering in the construction of P^* only states that are reachable in M_k for some k. The second example illustrates how a reachability assumption can be used to obtain a smaller closure for a very simple critical section problem. The transition graphs for M_0 and P in this example are shown in Figure 7-2. W is a *wait* state, and C corresponds to the *critical section*. This time we will show that the algorithm $M = \{M_0, M_1, \ldots\}$ is *2-reducible* or that $M_k\, C\, M_2$ for all $k \geq 2$. We choose as the closure of P the process shown in Figure 7-3. This is exactly what would be obtained by the construction described above except that transitions, which would result in states with more than one process in state c, have been eliminated. We use the same definitions for h_k and g_k as in the previous example. As before, it is easy to see that h_k and g_k satisfy the first two conditions in the definition of a homomorphism. It is also easy to establish the third condition provided we already know that only states with exactly one c component are reachable in M_k. This mutual exclusion property would, of course, have to be established by other techniques for proving safety properties or perhaps by the ICTL* decision procedure of Sistla and German [21]. Even when it is necessary to supply a reachability assumption of this sort, we believe our technique will still be useful for proving more complicated safety and liveness properties.

The reduction in the number of states of $M_r \times P^*$ obtained by using the reachability assumption is quite significant. Without the invariant $M_2 \times P^*$ has 24 states and $M_3 \times P^*$ has 48 states. With the reachability assumption, we only need to examine 5 states of $M_2 \times P^*$ and 6 states of $M_3 \times P^*$. Thus, with the reachability assumpiton it is relatively simple to show that $(M_2 \times P^*)\, D\, (M_3 \times P^*)$. It follows that M is *2-reducible* and that M_k and M_2 satisfy the same ICTL* formulas for every $k \geq 2$.

8. Conclusion

So far, we have only tried our procedure by hand on small examples. However, we expect to have completed a computer implementation in the near future. Obviously, our paper leaves open a number of important questions. Some are experimental in nature and can only be resolved by considering additional examples. For instance, how large is P^* in practice? Others questions, like the possibility of finding logics that are more expressive than ICTL*, require more theoretical work. In any case, it is clear that the claim in [2] regarding the infeasibility of automatically checking the correctness of programs with many processes was unduly pessimistic.

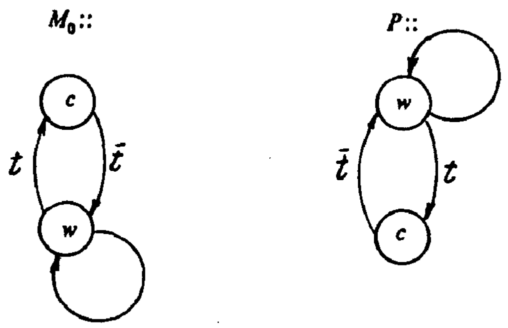

Figure 7-2: Critical Section Algorithm.

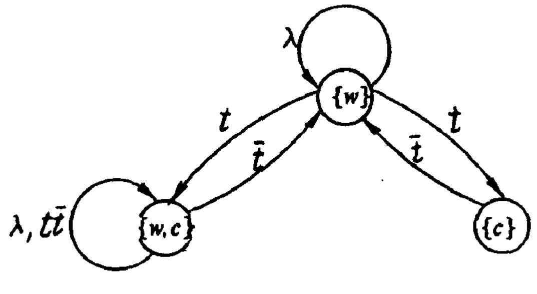

Figure 7-3: The Closure of P for the Critical Section Algorithm.

References

1. B. Alpern and F. Schneider. Verifying Temporal Properties without using Temporal Logic. Tech. Rept. 85-723, Cornell University Computer Science Department, December, 1985.

2. K. Apt and D. Kozen. "Limits for Automatic Verification of Finite-State Concurrent Systems". *Inf. Process. Lett. 22*, 6 (1986), 307-309.

3. M. C. Browne. An Improved Algorithm for the Automatic Verification of Finite State Systems using Temporal Logic. Proceedings of the 1986 Conference on Logic in Computer Science., Cambridge, Massachusetts, June, 1986, pp. 260-267.

4. M. Browne, E. Clarke, D. Dill, B. Mishra. "Automatic Verification of Sequential Circuits using Temporal Logic". *IEEE Transactions on Computers C-35*, 12 (December 1986).

5. M. C. Browne, E. M. Clarke, O. Grumberg. Characterizing Kripke Structures in Temporal Logic. Unpublished manuscript, submitted for publication.

6. E.M. Clarke, E.A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. Proc. of the Workshop on Logic of Programs, Yorktown Heights, NY, 1981.

7. E.M. Clarke, E.A. Emerson, A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems 8*, 2 (1986), 244-263.

8. E. M. Clarke, O. Grumberg, M. C. Browne. Reasoning about Networks with many identical finite-state processes. Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing., August, 1986, pp. 240-248.

9. David L. Dill and Edmund M. Clarke. "Automatic Verification of Asynchronous Circuits using Temporal Logic". *IEE Proceedings 133, pt. E*, 5 (September 1986).

10. E.A. Emerson and E.M. Clarke. Characterizing Properties of Parallel Programs as Fixpoints. Proc. of the Seventh International Colloquium on Automata, Languages and Programming, 1981.

11. E.A. Emerson, J.Y. Halpern. ""Sometimes" and "Not Never" Revisited: On Branching versus Linear Time". Proc. 10th ACM Symp. on Principles of Programming Languages, 1983.

12. E.A. Emerson, Chin Laung Lei. "Modalities for Model Checking: Branching Time Strikes Back". *Twelfth Symposium on Principles of Programming Languages, New Orleans, La.* (January 1985).

13. R. M. Karp and R. E. Miller. "Parallel Program Schemata". *JCSS* , 3 (1969), 147-195.

14. L. Lamport. What Good is Temporal Logic? Proceedings of the International Federation for Information Processing, 1983, pp. 657-668.

15. O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs Satisfy Their Linear Specification. Conference Record of the Twelth Annual ACM Symposium on Principles of Programming Languages, New Orleans, La., January, 1985.

16. R. Milner. *Lecture Notes in Computer Science.* Volume 92: *A Calculus of Communicating Systems.* Springer-Verlag, 1979.

17. B. Mishra, E.M. Clarke. "Hierarchical Verification of Asynchronous Circuits using Temporal Logic". *Theoretical Computer Science 38* (1985), 269-291.

18. C. A. Petri. Fundamentals of a theory of Asynchronous Information Flow. Proceedings of the IFIP Congress 62, Munich, 1962, pp. 386-390.

19. J.P. Quielle, J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". Proc. of the Fifth International Symposium in Programming, 1981.

20. A.P. Sistla, E.M. Clarke. "Complexity of Propositional Temporal Logics". *Journal of the Association for Computing Machinery 32*, 3 (J uly 1986), 733-749.

21. P. Sistla and S. German. Reasoning with Many Processes. GTE Laboratories Inc., Waltham, Massachusetts.

22. M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. Proceedings of the Conference on Logic in Computer Science, Boston, Mass., June, 1986.