THE MODEL CHECKING PROBLEM FOR CONCURRENT SYSTEMS WITH MANY SIMILAR PROCESSES

E.M. Clarke and O. Grumberg*
Computer Science Department
Carnegie Mellon University, Pittsburgh

1. INTRODUCTION

In [CGB] we addressed the problem of devising an appropriate logic for reasoning about concurrent systems 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* ([CES], [EC], [EH]) with the exception of the next-time operator and can, therefore, handle both linear and branching time properties with equal facility. 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 labeled 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.

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 affect the truth of a formula in our logic. We make this idea precise by introducing a new notion of equivalence (called ICTL*-equivalence) between Kripke structures with the same set of indexed propositions but different sets of index values. We prove 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. The most serious problem with this approach is that it requires an explicit representation of the state transition relations for the two Kripke structures. Sistla and German [SG] attempted to remedy this problem, but their approach had high complexity and would probably be quite difficult to implement.

In a later paper [CG], we showed how the explicit construction of the equivalence relation could be avoided in many cases. To understand how the new approach works, suppose that M_k has k identical copies of process P, i.e. $M_k = M_0 \times P^k$. Intuitively, we would like 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 does not quite work. It is possible to select M_0 and P in such a way that M_1 is ICTL*-equivalent

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

to M_2, but M_2 is not ICTL*-equivalent to M_3. Instead, we construct 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*. By using the CTL model checking algorithm on M_r, it should be possible to establish properties for M_k for all $k \geq r$. Thus, we are able to reduce an infinite set of verification problems to a single problem!

The present paper briefly surveys the results in [CGB] and [CG] and indicates how they might be used in reasoning about concurrent systems. It is organized as follows: Section 2 introduces the logics CTL and CTL* that we use for specifying finite state systems and briefly discusses the CTL model checking algorithm. Section 3 contains the definition of *stuttering equivalence*. The notation that we use for describing processes is presented in Section 4. Section 5 contains the definition of *ICTL** and discusses some restrictions on the logic that are necessary for systems with different numbers of similar processes to satisfy the same formulas. Process closures are introduced in Section 6 and the *collapsing theorem* is stated. Some examples of how the collapsing theorem can be used to avoid the state explosion problem are given in Section 7. The paper concludes in Section 8 with a discussion of some directions for future research.

2. THE LOGICS CTL AND CTL*

There are two types of formulas in CTL*: *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). Let AP be the set of atomic proposition names. A state formula is either:

- A, if $A \in AP$.
- If f and g are state formulas, then $\neg f$ and $f \vee g$ are state formulas.
- 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 $\neg f$, $f \vee g$, $\mathbf{X} f$, and $f \ \mathbf{U} \ g$ are path formulas.

CTL* is the set of state formulas generated by the above rules.
CTL is a subset of CTL* in which we restrict the path formulas to be:

- If f and g are state formulas, then $\mathbf{X} f$ and $f \ \mathbf{U} \ g$ are path formulas.
- If f is a path formula, then so is $\neg f$.

We define the semantics of both logics with respect to a Kripke structure $K = <S, R, L>$, where

- S is a set of states.
- $R \subseteq S \times S$ is the transition relation, which must be total. We write $s_1 \rightarrow s_2$ to indicate that $(s_1, s_2) \in R$.
- L: $S \rightarrow \mathbf{P}(AP)$ is the proposition labeling.

Unless otherwise stated, all of our results apply only to finite Kripke structures.

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, $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $s_i \rightarrow s_{i+1}$. π^i will denote the *suffix* of π starting at s_i.

We use the standard notation to indicate that a 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. $s \models A \quad \Leftrightarrow \quad A \in L(s)$.
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. $s \models \mathbf{E}(g_1) \quad \Leftrightarrow$ there exists a path π starting with s such that $\pi \models g_1$.
5. $\pi \models f_1 \quad \Leftrightarrow \quad s$ is the first state of π and $s \models f_1$.
6. $\pi \models \neg g_1 \quad \Leftrightarrow \quad \pi \not\models g_1$.
7. $\pi \models g_1 \vee g_2 \quad \Leftrightarrow \quad \pi \models g_1$ or $\pi \models g_2$.
8. $\pi \models \mathbf{X} g_1 \quad \Leftrightarrow \quad \pi^1 \models g_1$.
9. $\pi \models g_1 \mathbf{U} g_2 \quad \Leftrightarrow$ 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 will also use the following abbreviations in writing CTL* (and CTL) formulas:

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$
- $\mathbf{F} f \equiv \text{true } \mathbf{U} f$
- $\mathbf{A}(f) \equiv \neg \mathbf{E}(\neg f)$
- $\mathbf{G} f \equiv \neg \mathbf{F} \neg f$.

In [CES] automatic methods for *temporal logic model checking* were introduced. These techniques check that a finite-state concurrent system satisfies a CTL formula by searching all possible paths in the global state graph (Kripke structure) determined by the concurrent system. The following theorem states the complexity of the model checking algorithm.

Theorem 1: There is an algorithm for determining whether a CTL formula f_0 is true in state s of the structure $K = (S, R, L)$ that runs in time $O(length(f_0) \cdot (|S| + |R|))$.

Occasionally, we are only interested in the correctness of *fair* execution sequences in which some resource that is continuously requested by a process will eventually be granted to the process. This type of property cannot be expressed directly in CTL. In order to handle such properties we must modify the semantics of CTL in a such a way that the path quantifiers in CTL formulas will be restricted to fair paths. CTL with the new semantics is denoted by CTL^F. The following theorem shows that handling fairness in this manner does not change the complexity of the model checking algorithm.

Theorem 2: There is an algorithm for determining whether a CTL^F formula f_0 is true in a state s of a structure $K = (S, R, L, F)$ with F as the set of fairness constraints that runs in time

$O(length(f_0)\cdot(|S|+|R|)\cdot|F|)$.

3. EQUIVALENCE WITH RESPECT TO STUTTERING

We now define what it means for two Kripke structures to be equivalent with respect to stuttering. Given two structures K and K' with the same set of atomic propositions, we define a sequence of equivalence relations $C_0, C_1, \ldots$ on $S \times S'$ as follows:

- $s\ C_0 s'$ if and only if $L(s) = L(s')$.
- $s\ C_{n+1} s'$ if and only if
  1. for every path π in K that starts in s there is a path π' in K' that starts in s', a partition $B_1 B_2 \cdots$ of π, and a partition $B'_1, B'_2 \cdots$ of π' such that for all $j \in \mathbf{N}$, B_j and B'_j are both non-empty and finite, and every state in B_j is C_n-related to every state in B'_j, and
  2. For every path π' in K' starting in s' there is a path π in K starting in s that satisfies the same condition as in 1.

Our notion of *equivalence with respect to stuttering* is defined as follows: $s\ C\ s'$ if and only if $s\ C_i s'$ for all $i \geq 0$. Furthermore, we say that K with initial state s_0 is equivalent to K' with initial state s'_0 iff $s_0\ C\ s'_0$.

Lemma 3: Given two Kripke structures K and K', there exists an l such that $\forall s\ \forall s'[s\ C_l\ s'$ iff $s\ Cs']$.
Proof: By the definition of C_{l+1}, $s\ C_{l+1} s' \Rightarrow s\ C_l s'$, so $C_0 \supseteq C_1 \supseteq C_2 \cdots$. Since K and K' are both finite, C_0 must be finite as well, so only a finite number of these containments can be proper. Let C_l be the last relation that is properly included in C_{l-1}. By the definition of proper containment, $\forall\, m \geq l[C_l = C_m]$, so $s\ C_l s' \Rightarrow s\ C_m\ s'$, for all $m \geq l$. Since $s\ C_l\ s' \Rightarrow s\ C_{l-1} s' \Rightarrow s\ C_{l-2} s' \ldots$, we have $s\ C_l\ s' \Rightarrow \forall\, m[s\ C_m\ s']$, so $s\ C_l\ s' \Rightarrow s\ C\ s'$. The other direction is trivial. □

Theorem 4: If $s\ C\ s'$, then for every CTL* formula f without the next-time operator, $s \models f$ iff $s' \models f$. There exists a polynomial algorithm to compute C.

4. FINITE STATE PROCESSES

Next, we present a model of computation, suitable for reasoning about network of processes. Our model is similar to the CCS model used by Milner [M]. 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 synchronization, and *synchronization 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 $P = <AP, S, R, s_0, L>$ where,

- AP is the set of atomic propositions.

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

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

Let $P_1 = <AP_1,S_1,R_1,s_0^1,L_1>$ and $P_2 = <AP_2,S_2,R_2,s_0^2,L_2>$ be two processes. The *product process* $P_1 \times P_2 = <AP,S,R,s_0,L>$ 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 $s_1 = s'_1]$.
- $s_0 = (s_0^1,s_0^2)$.
- $L: S_1 \times S_2 \rightarrow \mathrm{P}(AP)$ 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 $(\ldots(P_1 \times P_2) \times \ldots P_{n-1}) \times P_n)$ 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 4.1, the product $M \times P$ is shown in Figure 4.2.

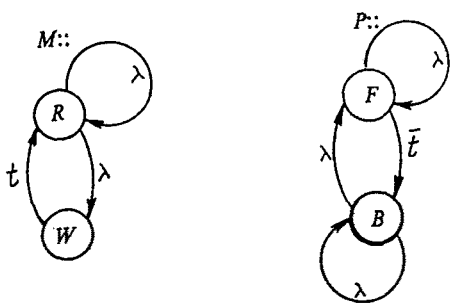

Figure 4.1: Two Finite State Processes: M and P.

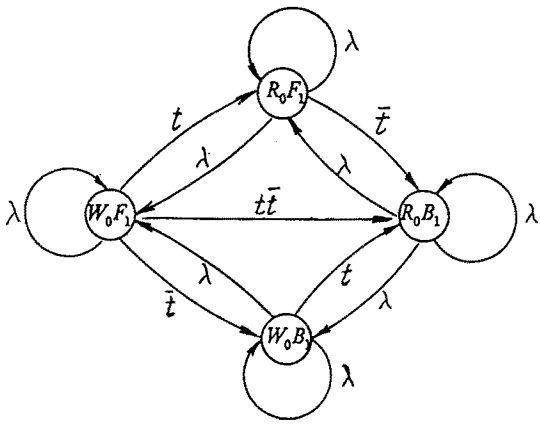

Figure 4.2: The Product $M \times P$.

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.

5. INDEXED CTL*

In order to reason about networks of processes, we need to be able to distinguish between the atomic propositions of the different processes. Therefore, we introduce the notion of *indexed atomic propositions* such that A_i is the value of proposition A in process i. Let AP be a set of proposition names which will be indexed by a set of index variables, IV. The logic *indexed* CTL* is an extension of CTL* where

- A_i is a state formula if $A \in AP$ and $i \in IV$.
- If f is a state formula that 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.)

Indexed CTL* is the set of *closed* state formulas generated by these rules and the rules in Section 2. We define the semantics of Indexed *CTL** with respect to a structure $K = \langle AP, I, S, R, s_0, L \rangle$, where AP, S, R, s_0 and L are defined as before and I is the set of index value (a subset of N). A *path in K* is also defined as before.

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. The relation $\models$ is defined as before, except that it satisfies also :

(1) $s \models A_i \Leftrightarrow A_i \in L(s)$.

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

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

We use $\bigwedge_i f(i)$ as an abbreviation for $\neg \bigvee_i \neg f(i)$.

We will omit the nexttime operator from indexed CTL*, 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,

$$\bigwedge_i \mathbf{A}(t_i \Rightarrow (\mathbf{XXX}t_i))$$

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.

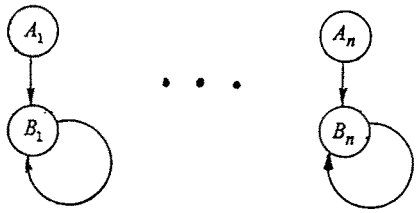

Figure 5.1: Example to Illustrate Restrictions on ICTL*

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 5.1. The following formula sets a lower bound on the number of processes:

$$\bigvee_i (A_i \wedge \mathbf{EF}(B_i \wedge \bigvee_j (A_j \wedge \mathbf{EF}(B_j \wedge \bigvee_k (A_k \ldots))))).$$

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 indexed *CTL**. The additional restrictions are:

- $\bigvee_i f$ is a permissible state formula only if f does not contain $\bigvee_j$ operators.
- $\bigvee_i g$ is not a permissible path formula.

- $g_1 \mathbf{U} g_2$ 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 can use the notion of equivalence defined in Section 3 to define an *indexed equivalence*. Since the restrictions to ICTL* do not permit the use of two different indices with an until operator, it is impossible to refer to the behavior of two different processes along a specific path. Thus, the notion of indexed equivalence between structures only needs to refer to one index from each structure at a time. Because of this, we will define a set of equivalence relations, $C_{ii'}$, that relate the behavior of an index i in one structure to the behavior of an index i' in the other structure.

Let K be a structure and i be an index value from I. The *reduction of K to i* (denoted by $K|_i$) is a structure identical to K except that the new proposition labeling L_i is defined as follows:

$$L_i(s) = \{A|A_i \in L(s)\}$$

In other words, all of the indexed atomic formulas are omitted except those that are indexed by i.

Let K_1 and K_2 be two structures with the same indexed atomic formulas and with index sets I_1 and I_2, respectively. We say that K_1 and K_2 (i,i') - correspond if and only if $K_1|_i \; C \; K_2|_{i'}$. We will write this as $K_1 C_{ii'} K_2$.
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 [CGB].

Theorem 5: If $K_1 C K_2$ then $K_1, s_0^1 \models h \Leftrightarrow K_2, s_0^2 \models h$, for every closed ICTL* formula h.

6. PROCESS CLOSURES

A distributed algorithm M is *r-reducible*, if and only if 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 6.1, 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 synchronization 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.

The homomorphisms g_k have similar properties, with $r+1$ replacing r.

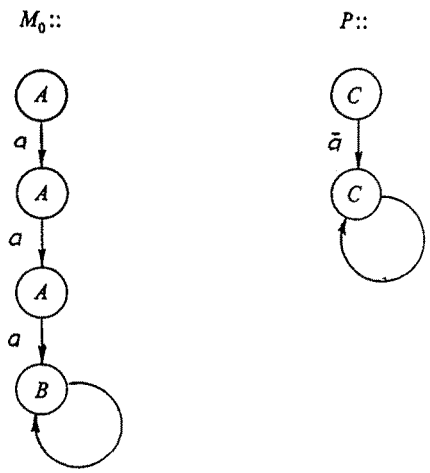

Figure 6.1: $M_1 \equiv M_2$, but $M_2 \not\equiv M_3$.

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 5. 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 5 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^*.

o For every path π', starting in σ_2, there exists a path π, starting in σ_1 that satisfies the same conditions above.

$E = \bigcap_n = E^n$, where E^n is defined exactly like D^n_{ii}, except that the basis case is given by $E^0 = \{(\sigma_1,\sigma_2) \mid p^*_1 = p^*_2\}$.

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

Theorem 6: If $M^P_r \, D \, M^P_{r+1}$ then for every $k \geq r$, $M_k \, C \, M_r$.
There exists a low-order polynomial algorithm to compute D.

7. EXAMPLES

To illustrate how the equivalence relation defined 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 4.1. 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 $M = \{M_1, M_2, ...\}$ with $M_k = M_0 \times P^k$ 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 $h_k : M_k \rightarrow M^P_1$ for $k \geq 1$ and $g_k : M_k \rightarrow M^P_2$ for $k \geq 2$ that satisfy the conditions given in Section 6.

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 of the processes are in the state F. The state $\{B\}$ represents a state of P^k in which all 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 $k > 0$.

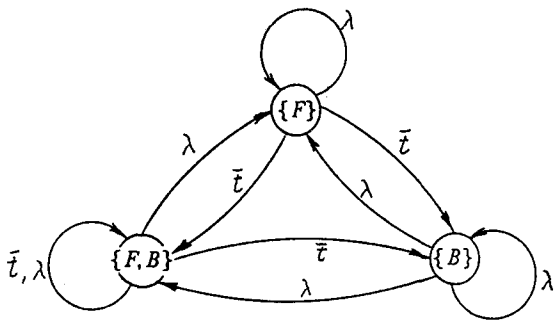

Figure 7.1: The Closure of P for the Master-Slave Algorithm.

The homomorphism h_k is also based on the intuition in the previous paragraph and is given by $h_k(s_0,...,s_r,s_{r+1},...,s_k) = (s_0,...,s_r,\{s_{r+1},...,s_k\})$. 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 to determine D 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 5 and Theorem 6, 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 [CES] 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: $P^* = <AP, S^*, R^*, s_0^*, L^*>$ where $S^* = \mathbf{P}(S) - \{\emptyset\}$. Intuitively, the states of P^* are abstractions of states of P^n. The state $\{s_1,...,s_k\}$ 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 $q = \{s_1,...,s_k\} \in S^*$. 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 $q' = (q - \{s_i, s_j\}) \cup \{s'_i, s'_j\}$ 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 labeling 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 in Figure 7.2. w is a *wait* state, and c corresponds to the *critical section*. This time we will show that the algorithm $M = \{M_1, M_2, \ldots\}$ 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 [SG]. 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.

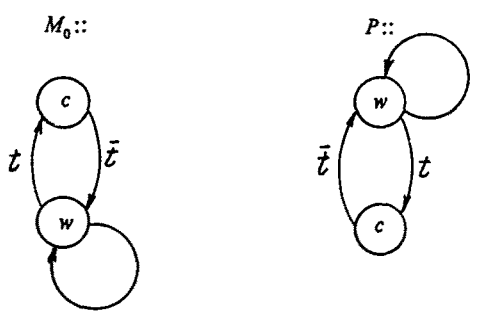

Figure 7.2: Critical Section algorithm

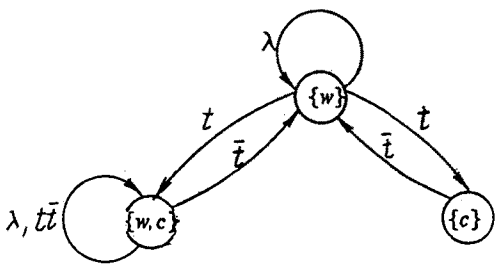

Figure 7.3: The Closure of P for the Critical Section Algorithm

The reduction in the number of states of $M_r \times P^*$ obtained by using the reachability assumption is quite significant. Without the variant $M_2 \times P^*$ 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 assumption 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. CONCLUSIONS

There are a number of important questions relating to *stuttering equivalence* that still need to be resolved. First and most importantly, what is the complexity of determining whether two states are stuttering equivalent? The best algorithm that we currently have is $O(n^4)$. We have implemented this algorithm on a VAX 11/780 and it seems to be relatively fast on small structures. In fact, we can handle a 100 state structure in 6 sec. We expect to be able to handle structures with several hundred states in reasonable time. The algorithm will probably not be useful for structures that are much larger than a thousand states. Since the goal of the theory described in this paper is to avoid building large structures, it is difficult to say whether the complexity of stuttering equivalence will prove to be a major limitation of our work. In any case, it would certainly be desirable to have a more efficient method for determining if two states are stuttering equivalent. Perhaps such a method exists for a restricted class of structures that is large enough to be useful in practice. Alternatively, there might be a weaker relation, which is easier to compute, that could be used in place of stuttering equivalence in many applications. *Observational equivalence* used in the study of CCS programs is closely related to stuttering equivalence and has somewhat lower complexity. It might be worthwhile to explore the relationship between these two notions of equivalence further.

There are also interesting open questions concerning the expressiveness of ICTL*. In Section 5 we showed how nesting of $\bigwedge_i$ and $\bigvee_i$ operators could be used to count the number of processes in a concurrent program. We conjecture in [CGB] that with formulas having at most k operators of this type, it is impossible to distinguish between programs that have more than k processes. In other words, if f is a formula with k levels of $\bigwedge_i$ and $\bigvee_i$ operators and M_n is a Kripke structure obtained as a product of n identical processes, then f will hold in M_n for $n > k$ if and only if f holds in M_k. If the conjecture is correct, it might be quite useful in verifying properties of systems with many processes. For example, mutual exclusion has nesting depth two so it may be sufficient to check this property for two process systems.

Finally, we have only tried the collapsing theorem on small examples. It is impossible to say how widely applicable this technique is without considering many more examples. Moreover, in order to make the technique practical we will need to investigate other network topologies (linear arrays, rings, etc.) and to develop procedures for finding process closures that have reasonable sizes.

REFERENCES

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

[CG] Clarke, E.M. and Grumberg, O., "Avoiding the State Explosion Problem in Temporal Logic Model Checking Algorithms", *Proc. of the 6th Annual ACM Symp. on Principles of Distributed Computing, ACM*, pp. 294-303, August 1987.

[CGB] Clarke, E.M., Grumberg, O., and Browne, M.C., "Reasoning about Networks with Many Identical Finite-State Processes", *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing, ACM*, pp. 240-248, August 1986. To appear in *Information and Computations*.

[EC] Emerson, E.A., and Clarke, E.M., "Characterizing Properties of Parallel Programs as Fix Points". Springer Lecture Notes in Computer Science. *Proc. of the 7th Intern. Colloq. on Automata Languages and Programming*, Vol. 85, Springer-Verlag, 1981.

[EH] Emerson, E.A., and Halpern, J.Y., '"Sometimes" and "Not Ever" Revisited: On Branching vs. Linear Time', *Proc. 10th ACM Symp. on Principles of Programming Languages*, 1983. And *J. of the ACM*, Vol. 33, No.1, January 1986, pp. 151-178.

[M] Milner, R., *Lecture Notes in Computer Science*, Vol. 92, *A Calculus of Communicating Systems*, Springer-Verlag, 1979.

[SG] Sistla, A.P., and German, S., "Reasoning with Many Processes", *Proc. of the Symp. on Logic in Computer Science*, Ithaca, N.Y., June 1987.