A Structural Induction Theorem for Processes

R. P. Kurshan
AT&T Bell Laboratories
Murray Hill, NJ 07974

K. L. McMillan
Carnegie Mellon University
Pittsburgh, PA 15213

March 9, 1993

Abstract

This paper deals with the formal verification of finite state systems that have an arbitrary number of identical components. We present a technique for inductively generalizing tests on a system of fixed size in order to show that a system of arbitrary size satisfies a given specification. This makes it possible to use finite state verification systems, such as COSPAN, to verify parameterized protocols. The method also may be useful for verifying systems of fixed but large size, since it reduces the size of the system that must be checked automatically. The basis of the method is a *structural induction theorem* for processes, which is stated and proved in this paper. The theorem applies to a variety of process formalisms satisfying simple algebraic laws. We give examples of proofs using the Calculus of Communicating Systems (CCS) and the s/r model.

1 Introduction

This paper deals with the formal verification of finite state systems that have an arbitrary number of identical components. Many protocols that occur in computers and communication are finite state, but are parameterized on the number of components in the system. For example, a network protocol may allow for an arbitrary number of hosts, or a system bus protocol may allow for an arbitrary number of processors. Since the structure of the system is inductively defined, some form of inductive reasoning is required to prove that the system satisfies a specification. We present a technique for inductively generalizing tests on a system of fixed size in order to show that a system of arbitrary size satisfies a given specification. This makes it possible to use finite state verification systems, such as COSPAN[1] [HK88, HK90] to verify parameterized protocols. The method may also be useful for systems of fixed size, since it

[1] COSPAN is a proprietary software system which may be licensed from AT&T by universities for educational purposes.

reduces the size of the system that must be checked automatically. This is significant because of the number of states of a system grows exponentially with the number of components. The basis of this method is a *structural induction theorem* for processes, which we state and prove in this paper.

A number of methods have been described previously for verifying properties of systems with an unbounded number of identical processes. Homomorphic reduction [Kur85, Kur87, Kur90] is a general framework for reducing the complexity of testing arbitrary ω-regular properties in finite-state systems. The regularity of systems of identical processes may facilitate the construction of a homomorphic reduction in such systems to a constant-size state space analysis (independent of the number of processes), as in [Kur85]. The analysis of the reduced system may be carried out automatically in time linear in the size of the reduced system. However, the construction of a homomorphic reduction is *ad hoc* and in systems of large or unbounded size is not automatically verifiable. Browne, Clarke and Grumberg [BCG86] introduced a method which uses an indexed form of branching-time temporal logic for specifications. Their method requires the *ad hoc* establishment of a form of bisimulation equivalence between global state graphs of systems of different size. This equivalence must be strong enough so that any two states in the same equivalence class satisfy the same set of formulas in the logic. In practice, this means that several restrictions must be placed on the logic. One of these restrictions is that the "next-time" operator is not allowed. Another restriction is that it is not possible to nest process quantifiers, with the consequence that some global system properties cannot be stated. A third method, due to German and Sistla [GS], uses a linear-time temporal logic for specifications (again, the next-time operator is not allowed). Their method is fully automatic (i.e., a bisimulation on structures of arbitrary size need not be established). By means of a distinguished "control" process, it is possible to check some global properties (although process quantifiers are not present in the logic). Unfortunately, the decision algorithm for this case is doubly exponential in the process size. Finally, methods similar to those described here can be found in [WL89, SG89].

The induction method presented here does not show satisfaction of a temporal logic formula by a model. Instead, both the implementation and specification are processes. Satisfaction of the specification is determined by a transitive reflexive relation on processes. That is, implementation p satisfies specification q if and only if $p \leq q$ in a suitable preorder. This preorder might be, for example, the language containment relation for ω-automata, as in [Kur85, Kur87] or one of the preorders for CCS processes[Hen88, Wal88]. Because the specification is a process, a verified specification may be used as an abstraction of the implementation for reasoning in a larger context. Also, there is no requirement that all systems, regardless of the size parameter, satisfy the same set of formulas in a logic. This makes the present method substantially more flexible than the methods mentioned above. In particular, "next-time" properties and global properties such as mutual exclusion are easily specified. The method is

not fully automatic, since an "invariant" process must be provided to carry out the induction step. Given this process, however, the checking procedure may be carried out automatically using the algorithm appropriate for the preorder. For example, if we use the language containment for s/r processes[Kur87], the complexity of this step is linear in the size of one system process, and quadratic in the size of the invariant process.

Application of the theorem is, in principle, quite simple. Once the system and its specification (or specifications) have been formalized using an appropriate process model, one contrives an invariant process. In the preorder, this invariant process must be not greater than the specification and not less than itself composed with the "next" process. Often an invariant process may be obtained by composing a small number of the system processes, then making minor modifications. In other cases, no modification is necessary; there exists some finite number of system processes which will serve as an invariant. Then again, there may be cases where a simple invariant process can be constructed based on an *ad hoc* abstraction of an arbitrary subset of the system processes. This paper contains examples of all three cases. With an invariant in hand, the induction theorem requires us to show three things. First, the "base case": the invariant must be not less than some finite number m of system processes. Second, the "induction step": the composition of the invariant with the $(m+1)^{th}$ system process is not greater than the invariant. Finally, the "satisfaction test": the invariant process must satisfy the specification. Of course, it is not necessary to guess a correct invariant at the first attempt. Since the above three checks may be carried out automatically, it is easy to experiment with different invariants until a correct one is found.

The following sections will make the above discussion precise, and provide some concrete examples as illustration. In section 2, we prove a safety property of a simple distributed algorithm, using Milner's CCS[Mil80] as the process model. We use the Concurrency Workbench[2] to prove three lemmas automatically, which we employ in a simple *ad hoc* proof by induction. This proof contains the basic ideas of induction on processes. In section 3, the structural induction theorem for processes is stated in greater generality and proved. The general theorem includes the notion of a "shift operator" on processes, which allows us to treat systems of processes with various communication topologies. In sections 4 and 5, we illustrate the various constructs introduced in the statement of the theorem, and the application of the theorem itself, by examining a simple algorithm for solving the "dining philosophers" problem, using the *s/r model* of coordinating processes[Kur85, Kur87] as a model. We use the COSPAN software system to prove the correctness of the algorithm automatically.

[2]The Concurrency Workbench is available from the Computer Science Department, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, Scotland.

2 A simple example of induction on processes

In this section, as an example, we consider an algorithm for maintaining the consistency of multiple copies of a data object in a distributed system. In this algorithm, any or all of the processors in the system may have a local copy of the object, but exactly one processor is considered the "owner" of the object[3]. The owner of an object maintains a list of the names of all processors which currently have a valid copy of the object. Any processor without a valid copy may receive a copy from the owner, although the number of processors which may have a copy at any one time may be limited. (We are not concerned here with finding the optimum number of copies here, but only with finding a correct algorithm which insures consistency.) Only the owner of an object is allowed to modify it. In order to maintain strict consistency, however, before modifying an object, the owner must invalidate all other copies of the object by sending an invalidate message to each processor with a valid copy. Finally, the ownership of the object may be transferred from one processor to another (assume the second has a valid copy already), by means of a special ownership-transfer message.

In order to simplify matters, we will assume that a data object is a single bit v which may be written with a one or a zero. We can consider each process to be in one of three conditions *vis-a-vis* the data object in question: *invalid* (Iv), *valid* (V), and *owner* (Ow). The *owner* state is parameterized by n, the number of additional processors which currently have copies. (We assume, of course, that the names of those processes are kept somewhere, but at this level of abstraction, only the number is of interest). The *owner* and *valid* states also are parameterized by v, the actual value of the data bit. The system has five kinds of messages: *read* messages (α_{rd}), *write* messages (α_{wr}), *copy* messages (α_{cp}), *invalidate* messages (α_{inv}), and *ownership transfer* messages (α_{ot}). Only read and write messages are visible to the programmer. A processor in the *invalid* state may receive a *copy* message, in which case it changes to the *valid* state. In the *valid* state, a processor may send *read* messages. This is intended to model a read of the data by a program. It may also receive an *invalidate* message, in which case it changes back to the *invalid* state, or it may receive an *ownership transfer* message, in which case it changes to the *owner* state. However, a processor in the valid state cannot receive write messages (modeling a write of the data by the program), since this may result in inconsistency of data copies. As the owner, a process may send *read* messages, or it may send out a *copy* message, in which case n is incremented by 1. The owner may also send out an *invalidate* message, in which case n is decremented by 1. If $n = 0$, the processor has the only valid copy of the object, and hence may receive a *write* message. For the purposes of this example, we will restrict n to be in the range $0 \leq n \leq 1$, *i.e.*, at most two processors may have a valid copy of the object.

[3]The simple protocol here is related to various cache consistency protocols. See [McM92] for a description of induction applied to a real cache consistency protocol.

We can formalize the above algorithm in a straight-forward way using Milner's CCS. Briefly, in CCS, if P is a process, then $\alpha.P$ is a process which evolves to process P by receiving a message α. The process $\overline{\alpha}.P$ evolves to P by sending a message α. The process $P \parallel Q$ is process P in parallel with process Q. In this composition, Q may receive a message sent by P or *vice versa*. This transmission appears to the outside world as an internal action represented by τ. The process $(P + Q)$ evolves to either P or Q nondeterministically. Finally $P \setminus \{\alpha\}$ restricts P so that it cannot send or receive message α.

A CCS description of our distributed data object is:

$$\begin{aligned}
Iv &= \alpha_{cp} v.V(v) \\
V(v) &= \alpha_{inv}.Iv + \alpha_{ot} n.Ow(n, v) + \overline{\alpha_{rd}} v.V(v) \\
Ow(n, v) &= \overline{\alpha_{rd}} v.Ow(n, v) \\
&\quad + (\underline{\text{if}}\ \ n = 0\ \ \underline{\text{then}}\ \ \alpha_{wr} v'.Ow(n, v') \\
&\qquad + \overline{\alpha_{cp}} v.Ow(n + 1, v) \\
&\qquad \underline{\text{else}}\ \ \overline{\alpha_{inv}}.Ow(n - 1, v) + \overline{\alpha_{ot}} n.V(v))
\end{aligned}$$

There are a variety of ways in which a CCS process could be viewed as a specification to be implemented. We will use a relation between processes called the "may" preorder. We say $S \leq_{may} T$, where S and T are CCS processes, if after any sequence of communications with the outside, T *may* send or receive any message that S *may* send or receive. In particular,

$$a.(b.c.nil + b.d.nil) \leq_{may} a.b.(c.nil + d.nil)$$

The reverse relation does not hold, however, since after sending $a.b$ the first process has already chosen to send c or to send d, while the second may still choose either. We might say that the "may" preorder allows the implementer to narrow down the choices provided by the specifier.

We will verify our distributed data object with respect to the simple process Bit, which we define below:

$$Bit(v) = \overline{\alpha_{rd}} v.Bit(v) + \alpha_{wr} v'.Bit(v')$$

This process has a parameter v representing a data value. It may send a read message containing its value, or it may receive a write message containing a new value. This process may be viewed as a specification,

In proving that the parallel composition of an arbitrary number of processors satisfies the specificaiton Bit, we assume that initially there is only one copy of the data object in the system, and that the object's value is $v = 0$. Our goal is, therefore, to prove

$$(Ow(0, 0) \parallel Iv \parallel \cdots \parallel Iv) \setminus \{\alpha_{cp}, \alpha_{ot}, \alpha_{inv}\} \leq_{may} Bit(0)$$

That is, for any number of processors, our distributed data object is less than or equal to Bit in the "may" preorder. Let's assume for the moment that we can find a process Q, such that the following inequalities hold:

$$\begin{aligned} Q \setminus \{\alpha_{cp}, \alpha_{ot}, \alpha_{inv}\} &\leq_{may} Bit(0) && (1)\\ Ow(0,0) &\leq_{may} Q && (2)\\ Q \parallel Iv &\leq_{may} Q && (3) \end{aligned}$$

If this is the case, we can use the fact that parallel composition is monotonic with respect to the "may" preorder to infer from 3 that

$$\begin{aligned} Q \parallel Iv &\leq_{may} Q\\ (Q \parallel Iv) \parallel Iv &\leq_{may} Q\\ ((Q \parallel Iv) \parallel Iv) \parallel Iv &\leq_{may} Q\\ &\vdots\\ Q \parallel Iv \parallel \cdots \parallel Iv &\leq_{may} Q \qquad (4) \end{aligned}$$

Again by monotonicity, we infer from 2 and 4 that

$$Ow(0,0) \parallel Iv \parallel \cdots \parallel Iv \leq_{may} Q$$

and hence,

$$\begin{aligned} &(Ow(0,0) \parallel Iv \parallel \cdots \parallel Iv) \setminus \{\alpha_{cp}, \alpha_{ot}, \alpha_{inv}\}\\ &\leq_{may} Q \setminus \{\alpha_{cp}, \alpha_{ot}, \alpha_{inv}\} \end{aligned}$$

since hiding is also monotonic with respect to the preorder. Thus, by transitivity of $\leq_{may}$, using (1), we conclude that

$$(Ow(0,0) \parallel Iv \parallel \cdots \parallel Iv) \setminus \{\alpha_{cp}, \alpha_{ot}, \alpha_{inv}\} \leq_{may} Bit(0)$$

The key to this simple proof is the process Q, which we will call a process invariant. The crucial property of Q is that it is greater than or equal to its composition with Iv. Finding such an invariant is closely analogous to finding a loop invariant in proving the correctness of a sequential program. There is no effective procedure for finding such invariants, but in practice, there are some useful techniques. The approach which we find successful in this case is to augment the system process slightly by freeing it to accept any observation after the occurrence of certain unexpected events.

For example, it is reasonable to suppose that $Ow(0,0)$ might be an invariant. We find, however, that $Ow(0,0) \parallel Iv \not\leq Ow(0,0)$. This is because $Ow(0,0) \parallel Iv$ may receive α_{cp}, but $Ow(0,0)$ may not. We may correct this problem by adding a term to $Ow(n,v)$. We note here that a collection of processes containing an $Ow(n,v)$ process does not expect to receive α_{cp} from its envoronment, since the number of $Ow(n,v)$ processes in the system is invariantly one. Therefore,

we will create a new process $Ow'(n,v)$ which is like $Ow(n,v)$, except that on receiving an α_{cp} message, it evolves to a process T which allows any observation. We are saying, in effect, that the invariant puts no restrictions on what may happen after this "illegal" message is received.

Carrying on on this way, we arrive at the following version of Q:

$$\begin{aligned}
Q &= Ow'(0,0) \\
Iv' &= \alpha_{cp} v.V'(v) \\
V'(v) &= \alpha_{inv}.Iv' + \alpha_{ot} n.Ow'(n,v) \\
&\qquad + \overline{\alpha_{rd}} v.V'(v) + \alpha_{cp}(v').T \\
Ow'(n,v) &= \overline{\alpha_{rd}} v.Ow'(n,v) \\
&\quad + \alpha_{cp}(v').T + \alpha_{ot}[n'].T + \alpha_{inv}.T \\
&\quad + (\underline{\text{if}}\ n = 0\ \underline{\text{then}}\ \alpha_{wr} v'.Ow'(n,v') \\
&\qquad + \overline{\alpha_{cp}} v.Ow'(n+1,v) \\
&\qquad + \overline{\alpha_{ot}}(1).T + \overline{\alpha_{inv}}.T \\
&\quad \underline{\text{else}}\ \overline{\alpha_{inv}}.Ow'(n-1,v) + \overline{\alpha_{ot}} v.V'(v)) \\
T &= \overline{\alpha_{rd}} v.T + \alpha_{wr} v.T + \alpha_{cp} v.T + \overline{\alpha_{cp}} v.T \\
&\quad + \alpha_{inv}.T + \overline{\alpha_{inv}}.T + \alpha_{ot}(1).T \\
&\quad + \overline{\alpha_{ot}}(1).T
\end{aligned}$$

Note that if "internal" actions α_{cp}, α_{ot} and α_{inv} are restricted, it is not possible for Q to evolve to T. In fact, $Q \setminus \{\alpha_{cp}, \alpha_{ot}, \alpha_{inv}\}$ is observationally equivalent to $Bit(0)$. Thus, condition (1) is satisfied. By construction, condition (2) also is satisfied. To understand why this choice of Q also satisfies condition (3), it is useful to think of Q as representing some arbitrary subset of the processors in our system. In this view, n represents the number of processors outside the subset which have valid copies. Consider now adding one more process to our subset, i.e., let the initial state be $Q \parallel Iv$. This process may receive an α_{cp} message, since it contains a process in the *invalid* state. To satisfy condition (3), Q must also accept this message. On the other hand, since $Q = Ow'(0,0)$ represents a state where the owner of the object is inside the subset, we know this message will never be received from outside the subset, since the number of owners is invariantly one. Thus, it is safe to allow any behavior after receiving such a message, and we add a transition $\alpha_{cp} v.T$. To complete the definition, we define a map ρ from the reachable states of $Q \parallel Iv$ to the reachable states of Q. (Here, "reachable" means reachable assuming only correct behaviors of the processes outside the subset). This map represents, in some sense, the abstraction by which we represent the behavior of many processes by one process. It has the property that if $A \xrightarrow{\alpha} B$, then either $\rho(A) \xrightarrow{\alpha} \rho(B)$ or $\rho(A) \xrightarrow{\alpha} T$. The definition of ρ is as follows:

$$\rho^{-1}(Ow'(0,v)) = \{(Ow'(0,v) \parallel Iv'), (Ow'(1,v) \parallel V'(v)),$$

$$
\begin{aligned}
 & (Iv' \parallel Ow'(0,v)), (V'(v) \parallel Ow'(1,v))\} \\
\rho^{-1}(Ow'(1,v)) &= \{(Ow'(1,v) \parallel Iv'), (Iv' \parallel Ow'(1,v))\} \\
\rho^{-1}(V'(v)) &= \{(V'(v) \parallel Iv'), (Iv' \parallel V'(v))\} \\
\rho^{-1}(Iv') &= \{Iv' \parallel Iv'\}
\end{aligned}
$$

We complete the definition of Q in the following way, using this map. If any state on the right hand side of the above equations can accept an action which is not accepted on the left hand side, we add a transition to T on that action to the state on the left hand side. Of course, this invariant is not intended to be "correct by construction". Instead, we used the Concurrency Workbench[4] to prove automatically that propositions (1) and (2), and (3) are satisfied. Thus, using an automatic verifier and some simple inductive reasoning, we are able to construct a proof of correctness for a system with an arbitrary number of processes.

3 The structural induction theorem for processes

In this section, we set up a general framework for applying structural induction to a system of concurrent processes. This framework is independent of any particular process model, or process ordering. Provided the model and the ordering satisfy some simple conditions, it allows us to prove a general induction theorem. We begin with a class of "processes" $\mathcal{P}$ which, for our present purposes, may be elements of an arbitrary set. On this domain, we define a binary operator $\otimes$, a unary operator ϕ, and a preorder $\leq$, all on $\mathcal{P}$. The operator $\otimes$ produces the (parallel) composition of two processes, which is a process in $\mathcal{P}$. A "system" process composed of an array of processes $p_1, p_2, ..., p_n$ is represented as $p_1 \otimes p_2 \otimes \ldots \otimes p_n$, which we abbreviate to $\bigotimes_{i=1}^{n} p_i$. We do not assume that $\otimes$ is associative. Rather, we use the convention that left-most arguments bind first if parentheses are omitted.

The unary operator $\phi : \mathcal{P} \rightarrow \mathcal{P}$ is called the "shift" operator. It maps process p_i to the process p_{i+1} $(i = 1, 2, ..)$, usually through some kind of parameter substitution. In our previous example, ϕ was the identity, since there was no distinction between two processes in the system. In general, however, the processes in a system will not be syntactically indistinguishable. Rather, process p_i will have i as a parameter. The parameter might determine, for example, which communication channels p_i sends and receives messages over. To make an inductive argument about such a collection of processes, we need to know how process p_{i+1} is derived from process p_i. This is the purpose of the function ϕ. The fact that ϕ is defined over individual system processes and not systems should make it clear that we are not arguing by induction over the total size of

[4] The input text for the example is available from the second author.

the system, but rather over the subsets of system processes of increasing size. This is a fairly subtle distinction, but it makes possible reasoning about closed rings of an arbitrary number of processes, as we will observe later.

The preorder in the previous example was the "may" preorder on CCS processes [Hen88]. There are other such preorders however, including the language containment relation for s/r-processes [Kur87] and the conformance relation for finite trace structures [Dil88].

We make the following requirements on $\otimes$, ϕ and $\leq$:

- The composition operator $\otimes$ must be monotonic with respect to $\leq$, *ie.*,

$$\forall p, q, r \in \mathcal{P} : p \leq q \Rightarrow p \otimes r \leq q \otimes r \tag{5}$$

- The operator ϕ must distribute over $\otimes$, *ie.*,

$$\forall p, q \in \mathcal{P} : \phi(p \otimes q) = \phi(p) \otimes \phi(q) \tag{6}$$

- The operator ϕ must preserve the relation $\leq$, *ie.*,

$$\forall p, q \in \mathcal{P} : p \leq q \Rightarrow \phi(p) \leq \phi(q) \tag{7}$$

The *structural induction theorem* now can be stated as follows:

Theorem 1 *Let $\{p_1, p_2, \ldots, p_n\}$ and $\{q_1, q_2, \ldots, q_n\}$ be two sets of processes in $\mathcal{P}$. Let m be an integer $1 \leq m < n$ such that for all $m < i < n$, $p_{i+1} = \phi(p_i)$ and for all $m \leq i < n$, $q_{i+1} = \phi(q_i)$.*

If

$$\bigotimes_{i=1}^{m} p_i \leq q_m \tag{8}$$

and

$$q_m \otimes p_{m+1} \leq q_{m+1} \tag{9}$$

then for all j, $m \leq j \leq n$,

$$\bigotimes_{i=1}^{j} p_i \leq q_j. \tag{10}$$

Proof: If $n = m + 1$ then (10) follows from (5), (8) and (9) directly. Otherwise, applying properties (6) and (7) of ϕ to proposition (9) above, we have

$$\phi(q_m) \otimes \phi(p_{m+1}) \leq \phi(q_{m+1}) \tag{11}$$

and $p_{m+2} = \phi(p_{m+1})$, $q_{m+1} = \phi(q_m)$ and $q_{m+2} = \phi(q_{m+1})$. Applying ϕ to (9), get

$$q_{m+1} \otimes p_{m+2} \leq q_{m+2} \tag{12}$$

and, by repeated application of the above two steps,

$$\forall i, m \leq i < n : q_i \otimes p_{i+1} \leq q_{i+1}. \tag{13}$$

Using the monotonicity property of $\otimes$, we compose both sides of (8) with p_{m+1} to obtain

$$\bigotimes_{i=1}^{m+1} p_i \leq q_m \otimes p_{m+1} \tag{14}$$

which, using (13) and the transitivity of $\leq$ gives us

$$\bigotimes_{i=1}^{m+1} p_i \leq q_{m+1}. \tag{15}$$

Again, by repeated application of these two steps,

$$\forall j, m \leq j \leq n : \bigotimes_{i=1}^{j} p_i \leq q_j \tag{16}$$

□

The significance of theorem 1 is that the antecedents, conditions (8) and (9), make no reference to n, the number of processes in the array. Thus, by proving two propositions about arrays of fixed size m, we may draw a conclusion about an array of arbitrary size n. In general, given an array $P = \bigotimes_{i=1}^{n} p_i$, and a specification (or task) T, we can use the theorem to prove some statement of the form $P \leq T$, or $P' \leq T$, where P' is some order-preserving restriction or projection of P. In order to do this, we need the following elements: the process invariant q_i, the shift operator ϕ, and a method of automatically verifying conditions (8) and (9). Given these, we will be able to replace the problem of verifying $P \leq T$ or $P' \leq T$ with verifying $q_n \leq T$ or $q'_n \leq T$. In the cases of interest, T is independent of n, q_n has a fixed size independent of n, and these tests are straight-forward.

We should note that although the automatic checks require only a single invariant process q_m, the proof of the theorem involves a sequence of processes $q_m \ldots q_n$. The reason for using this sequence will become apparent when we consider ring structured systems. In this case we would like the invariant to apply to the behavior visible at the ends of a string of processes of arbitrary length. Since the "end" process is parameterized on i, the invariant must also be parameterized on i. Though the invariant is parameterized, we can prove (8) and (9) using only a fixed parameter m.

4 Induction and the s/r model

As an illustration of the various components of the induction theorem, we will apply the theorem using a process formalism which is different from CCS in

many respects: the s/r model [Kur87, Kur90] of concurrent processes. Our example will be an algorithm for solving the dining philosophers problem of E.W. Dijkstra. This algorithm has been previously analyzed using an *ad hoc* homomorphic reduction [Kur85]. The reduction in this analysis was not automatically checked. In the proof presented here, we need only prove by hand that the preconditions of the induction theorem are met. This can be done once for many examples with the same basic structure. The preorder tests (8) and (9) can be made automatically using the COSPAN system [HK90]. Using the s/r model, we show how "eventuality" or "liveness" properties may be proved using process induction, and also how the induction theorem may be applied to systems with a ring-like structure.

An L-process p has four components (S, M, C, I) where

- S is the set of states.
- $M : S \times S \rightarrow L$ is a matrix over a Boolean algebra L. Intuitively, the matrix element $M(i, j)$ is the condition under which the process may evolve from state i to state j.
- $C \subseteq 2^S$ is a set of subsets of the states called *cycle sets*. Cycle sets exclude certain infinite behaviors (see below).
- $I \in S$ is the initial state.

A Boolean algebra has operators $\wedge$, $\vee$ and $\neg$. Every Boolean algebra is isomorphic to a power set (all subsets of a given set) where $\wedge$ is intersection, $\vee$ is union, and $\neg$ is complement. The singletons in this power set are called *atoms*. In the s/r model, there is a collection of variables called *selections*, and an atom is an assignment of a value to every selection variable. Selections are the means of communication between processes.

Every L-process p has a language $\mathcal{L}(p)$. The language is a set of ω sequences of atoms of L. In particular, any infinite sequence $a_0, a_1, \ldots$ of atoms of L is in $\mathcal{L}(p)$ exactly when there exists a sequence of states $s_0, s_1, \ldots$ such that for all $i \in \omega$, $a_i \in M(s_i, s_{i+1})$, and $s_0 = I$. However, the set of states occurring infinitely in $s_0, s_1, \ldots$ may not be contained in any *cycle set* (element of C). Cycle sets are useful when we wish to assume that a given process will not stay in a given state or states infinitely. This allows us to prove liveness properties about L-processes.

Language containment is a suitable preorder for the induction theorem. Let $p_1 \leq p_2$ if and only if $\mathcal{L}(p_1) \subseteq \mathcal{L}(p_2)$. This relation is clearly reflexive and transitive, since $\subseteq$ has these properties.

The parallel composition of two L-processes p_1 and p_2 is denoted $p_1 \otimes p_2$. This notation derives from the fact that the transition matrix associated with the parallel compositon is the tensor product of the matrices associated with p_1 and p_2. To be exact, if $p = p_1 \otimes p_2$ then

- $S = S_1 \times S_2$
- $M((i_1, i_2), (j_1, j_2)) = M_1(i_1, j_1) \wedge M_2(i_2, j_2)$
- $C = \{c \times S_2 \mid c \in C_1\} \cup \{S_1 \times c \mid c \in C_2\}$
- $I = (I_1, I_2)$

Note that the cycle sets of the composition are formed by lifting the cycles sets of p_1 and p_2 into the product state space. Thus a state sequence in the composition is legal when neither subprocess remains infinitely in one of its cycle sets. We can show that

$$\mathcal{L}(p_1 \otimes p_2) = \mathcal{L}(p_1) \cap \mathcal{L}(p_2)$$

Since parallel composition is equivalent to language intersection, and intersection is monotonic with respect to set containment, it follows that composition is monotonic with respect to language containment. Hence, $\otimes$ is a suitable composition operator for the induction theorem.

A homomorphism on a Boolean algebra L is any map $\phi : L \rightarrow L$ such that $\phi(x \wedge y) = \phi(x) \wedge \phi(y)$. We can apply such a map to an L-process by simply applying it to every element of the transition matrix. Thus if $p = \phi(p')$, then let $M(i, j) = \phi(M'(i, j))$.

As an example, suppose we have two selection variables a and b. Every atom in our Boolean algebra can be expressed as $(a = i) \wedge (b = j)$ for some i and j. Now suppose the function ϕ takes every occurrence of b and changes it to a. Then, for example,

$$\phi((a = 1) \wedge (b = 2)) = (a = 1) \wedge (a = 2) = \text{false}.$$

On the other hand,

$$\phi((a = 1) \wedge (b = 1)) = (a = 1) \wedge (a = 1) = (a = 1).$$

When we apply ϕ to an L-automaton, we get another L-automaton with a different language over the same Boolean algebra. Note, however, that there is not a one-to-one correspondence between sequences in these two languages. For example, any sequence in the old language containing the atom $(a = 1) \wedge (b = 2)$ corresponds to no strings in the new language, since this atom maps to false. On the other hand, a sequence containing the atom $(a = 1) \wedge (b = 1)$ may correspond to many sequences in the new language, since this atom maps to the non-atom $(a = 1)$, which contains many atoms with different valuations of b.

We observe from the above example that a homomorphism does not necessarily map atoms onto atoms. Homomorphisms are, however, monotonic. That is, from $\phi(x \wedge y) = \phi(x) \wedge \phi(y)$, it follows that $x \subseteq y$ implies $\phi(x) \subseteq \phi(y)$. From this it follows that $\mathcal{L}(M) \subseteq \mathcal{L}(M')$ implies $\mathcal{L}(\phi M) \subseteq \mathcal{L}(\phi M')$. That is, Boolean algebra homomorphisms are monotonic with respect to language containment.

The Boolean algebra homomorphisms that we use are all component-wise homomorphisms, in the sense that they derive from replacing each selection variable by some function of the selection variables. For example, if the selection variables are $s_0, \ldots s_3$, then

$$s_i \leftarrow s_{i+1} \bmod 4$$

describes a homomorphism that shifts the variables in a ring, replacing each selection by its neighbor to the "right". Another valid homomorphism would be

$$s_i \leftarrow s_i + s_{i+1} \bmod 4$$

In this case we replace every selection variable by its sum with its neighbor to the right. However, in this paper, we require homomorphisms that satisfy the conditions in the induction theorem for the shift operator. This will allow us to "shift" our invariant q_i, decribing a property of a group of i processes, to q_{i+1}, describing a property of a group of $i+1$ processes. For this to occur, it is necessary that $\phi(p_i) = p_{i+1}$. The exact nature of ϕ will therefore depend on the structure of the system of processes, and in particular, how the parameter i appears in p_i. In the sequel, we will describe an appropriate shift operator for a ring structured system.

5 Dining philosophers

We assume the reader is familiar with Dijkstra's famous Dining Philosopher's paradigm. We which present an implementation prevents deadlock, and hence starvation of the philosophers, by breaking the symmetry of the network. This is accomplished through introduction of a "negative" token (the "encyclopedia"), which is passed around the ring of philosophers [Kur85]. The rules of the game are as follows. Each philosopher is in one of three states: thinking, eating, or reading (the encyclopedia). A thinking philosopher who becomes hungry may begin eating if neither of his neighbors is currently eating, and if his neighbor to the left is not hungry. However, if he is not hungry, and his neighbor to the left is currently reading, he must take the encyclopedia and enter the reading state.

Our formal model of the algorithm for n dining philosophers is the L-process $P = \bigotimes_{i=1}^{n} p_i$, where p_i is as defined below. Each process p_i has three states: **THINK**, **EAT** and **READ**. The initial state of p_i is **THINK** for $i > 1$ and **READ** for $i = 1$. Thus the encyclopedia starts at philosopher 1, breaking the symmetry. The Boolean algebra L is defined in terms of n selection variables, $x_1 \ldots x_n$, all of which have the domain $\{think, hungry, eat, read\}$. The process model for p_i is defined in the S/R language [KK86] in figure 1: a process is formed by instantiating a `proctype` through substitution of selection variables $x_1, x_2, \ldots$ in place of the formal parameters of type *selection* in the `proctype`.

The `trans` statement defines the allowable values of the local selection variable for each state as a name-list within braces, and defines the conditions for making a transition from one state to another. For example, in the **THINK** state, a philosopher process p_i may select either $x_i = \mathit{think}$ or $x_i = \mathit{hungry}$. It makes a transition to the **EAT** state if $x_i = \mathit{hungry}$ and it is not blocked, where blocking is defined by a macro to be the condition where either neighbor is selecting *eat* (*ie.*, holding both forks) or the left neighbor is selecting *hungry* (*ie.*, philosophers defer to their left). (Although we do not discuss it further here, this algorithm may be symmetrized by introducing n initial states into the "system" process P of n philosophers: for $1 \leq i \leq n$, the i^{th} initial state of P has p_i in the **READ** state, and all the others in the **THINK** state. The same analysis we give here goes through in this symmetrized case as well.)

Each process p_i has a single cycle set, {**EAT**}, which represents the assumption that no philosopher will remain in the eating state forever (a Weight-Watcher convention).

We prove first that that the first philosopher p_1 does not starve, for any n. Then we prove that for all $1 \leq k \leq n$, philosopher p_k does not starve. We assume that $n \geq 4$. The cases $1 \leq n \leq 3$ can be checked explicitly using COSPAN. The condition that p_1 does not starve is represented by the language of the "task" process T_i, defined in S/R in figure 2 (for input parameter $x = x_i$). The language of T_i is the set of all sequences consistent with the transition structure of T_i, except those in which the selection of x eventually is constantly *hungry*. This language is defined by a two state process, which enters state 1 when the selection $x_1 = \mathit{hungry}$, and state 0 otherwise. The cycle set {1} excepts from the language all sequences which cause the process eventually to remain in state 1.

We choose ϕ to be the following Boolean algebra homomorphism:

- $x_1 \leftarrow x_1$
- $x_i \leftarrow x_{i+1}$, for all $1 < i < n$
- $x_n \leftarrow x_1$

This map shifts all of the variables up by one, except x_1; the variable x_n wraps around to x_1.

The process invariant q_i defined to be $p_{i-1} \otimes p_i \otimes r_i$, where r_i is the two-state process of figure 3. This process guarantees that if p_1 selects *read*, then eventually p_{i-1} will select *read*.

For all $4 < i < n$, $p_{i+1} = \phi(p_i)$ and for all $4 \leq i < n$, $q_{i+1} = \phi(q_i)$, which can be verified symbolically by COSPAN. Hence, we can apply theorem 1 for $m = 4$. Using COSPAN, we automatically verify the following assertions:

$$\mathcal{L}(\bigotimes_{i=1}^{4} p_i) \subseteq \mathcal{L}(q_4), \tag{17}$$

$$\mathcal{L}(q_4 \otimes p_5) \subseteq \mathcal{L}(q_5). \tag{18}$$

```
type selection :  (think,hungry,eat,read)
type state : (THINK,EAT,READ)

proctype p(x_i_minus_1, x_i_plus_1 : selection; initial : state)
  import x_i_minus_1, x_i_plus_1
  stvar $ : state
  selvar x : selection
  init initial
  macro Blocked := (x_i_minus_1: hungry,eat) + (x_i_plus_1: eat)
  cyset {EAT}
  trans
    THINK {think,hungry}
      -> EAT   : (x = hungry) * ~Blocked
      -> READ  : (x = think) * (x_i_minus_1 = read)
      -> THINK : else;
    EAT {eat, think}
      -> THINK : (x = think) * (x_i_minus_1 ~= read)
      -> READ  : (x = think) * (x_i_minus_1 = read)
      -> EAT   : else;
    READ {read}
      -> THINK : x_i_plus_1 = think
      -> READ  : else;
end
```

Figure 1: The philosopher process p_i.

Thus, by theorem 1, we conclude that

$$\mathcal{L}(\bigotimes_{i=1}^{n} p_i) \subseteq \mathcal{L}(q_n). \tag{19}$$

This proof is very close in structure to the proof one might write by hand. In this case, one would argue by induction starting with some small number of processors as the base case. Here, the base case is, if p_1 is reading, then eventually p_4 will read. This corresponds to (17). The induction step in our hand proof would be that if (p_1 is reading) $\Rightarrow$ (p_i will read), then (p_1 is reading) $\Rightarrow$ (p_{i+1} will read). This corresponds roughly to (18). Theorem 1 tells us we only need to test the case ($i = 4$). In the induciton step, to prove something about p_{i+1}, we rely on the behavior of philosophers p_{i-1}, p_i. Thus, our hand proof of the invariant would make reference to philosophers p_{i-1} and p_i. Similarly, our invariant process q_i includes processes p_i and p_{i-1}.

The proof of the induction step would run roughly as follows: If philosopher

```
proctype T(x_i : selection)
  import x_i
  stvar $ : (0,1)
  init 0
  cyset {1}
  trans
    0,1
      -> 1 : x_i = hungry
      -> 0 : else;
end
```

Figure 2: The task process T_i.

p_{i-1} eventually has the encyclopedia, then since philosopher p_{i+1} cannot stay in the eating state forever, eventually philosopher p_i becomes unblocked. If philosopher p_i becomes unblocked, it eventually reaches the thinking state, and hence takes the encyclopedia from philosopher p_{i-1}, entering the reading state. Although this is a fairly vague argument, there is no need to formalize it, since the induction step is checked automatically by COSPAN. An informal understanding of this proof is necessary in order to construct the invariant process q_i. In particular, one must understand the protocol well enough to realize that philosophers p_{i-1} and p_i are necessary to the induction step, and thus must be included in q_i. However, one can obtain this information by trial and error, since COSPAN provides behavioral traces as counterexamples if the induction step fails. This is in fact how the invariant presented here was obtained.

We complete the verification that philosopher p_1 never starves, as follows. Using COSPAN, we check

$$\mathcal{L}(p_1 \otimes p_2 \otimes q_n) \subseteq \mathcal{L}(T_1).$$

(It is necessary to include philosopher p_2 in order to prevent p_1 from being blocked indefinitely, which would be the case if selection ($x_2 = eat$) were true indefinitely.) From the result of the induction (19) and monotonicity of $\otimes$, we conclude that

$$\mathcal{L}(\bigotimes_{i=1}^{n} p_i) \subseteq \mathcal{L}(T_1). \qquad \Box \tag{20}$$

Finally, we prove that p_k does not starve for all $1 \leq k \leq i$. This result cannot be obtained purely by symmetry of the ring, because the fact that p_1 initially has the encyclopedia breaks the symmetry. Instead, we check the following relation using COSPAN:

$$\mathcal{L}(p_1 \otimes p_2 \otimes q_k \otimes p_{k+1} \otimes q_n) \subseteq \mathcal{L}(T_k).$$

```
proctype r(x_1,x_i_minus_1 : selection)
  import x_1,x_i_minus_1
  stvar $ : (0,1)
  init 0
  cyset {1}
  trans
    0
      -> 1 : (x_1 = read) * (x_i_minus_1 ~= read)
      -> 0 : else;
    1
      -> 0 : (x_i_minus_1 = read)
      -> 1 : else;
end
```

Figure 3: The process r_i

We already have proved by theorem 1 that for $4 \leq k \leq n$, $\bigotimes_{i=1}^{k} p_i \leq q_k$. Hence, by monotonicity,

$$\mathcal{L}(\bigotimes_{i=1}^{n} p_i) \subseteq \mathcal{L}(T_k). \tag{21}$$

The cases $2 \leq k < 4$ can be verified explicitly using COSPAN, in the same way that the case $k = 1$ was verified.

This proof works because p_1, p_2 and q_n are sufficient to guarantee that p_1 is reading infinitely often. This and q_k guarantee that p_{k-1} is reading infinitely often. Along with the fact that p_{k+1} does not eat forever, this is enough to show that p_k does not starve.

By using COSPAN[5] to check relations between fixed processes with no free parameters, we have proved a property of a parameterized system of n processes. This proof is automatically checked to the following extent. The conclusion (21) can be stated formally in the syntax of the S/R language. In principle, it could be verified automatically by COSPAN for any fixed value of the parameter n, but not simultaneously for all n. In order to prove (21) simultaneously for all n, we have presented three checks involving a fixed number of processes, which are verified by COSPAN. Then, by the induction theorem, we concluded in effect that (21) would be verified by COSPAN for any given value of n (presuming COSPAN could deal with the computational complexity associated with that n). This conclusion is based on the reasoning presented in this paper, and this reasoning is not checked by COSPAN (although it could theoretically be checked in a theorem-prover interfaced with COSPAN).

[5]The input text for the example is available upon request.

There is an interesting alternative process invariant for this problem which was discovered by Peter Ramage. It turns out that the induction carries through if we simply let $q_7 = \bigotimes_{i=1}^{7} p_i$, projecting the language onto the state variables x_n, x_1, x_7, x_8. In other words, viewed from the ends, a segment of 7 philosophers is indistinguishable from 8 philosophers. This property holds for no number less than 7. Although we do not know the reason for this, it illustrates a very simple approach to generating process invariants.

6 Summary

We have seen, in two specific cases, how the problem of verifying a protocol involving an unbounded number of processes in a regular organization can be solved using an automatic verification system and a simple induction theorem. The theorem may be applied to any process formalism, provided it satisfies the conditions set forth in section 3. It reduces a problem involving a parameterized system of n processes to a problem involving a fixed collection of processes with no parameters. As a result, the proof is somewhat different from an ordinary induction proof, since instead of showing that q_i implies q_{i+1}, we show only, for example, that q_4 implies q_5. This approach relies on all of the q_i and p_i being identical, except for being "shifted" by an appropriate process map. Our examples represent two different communication structures: a fully connected network, and a ring. The theorem is general enough to apply to other structures; in fact, it can be generalized to any inductively defined structures. In addition, the method is also more general than previously published methods in the range of properties that can be proved, since there is no requirement that all formulas in a (restricted) temporal logic be preserved between systems of varying numbers of processes.

7 Acknowledgements

This paper benefited from some lengthy discussions with David Long at Carnegie Mellon University. The CCS example was inspired by a talk given at CMU by John Hennessy.

References

[BCG86] M.C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. In *ACM Symp. Principles of Distributed Computing 5*, 1986.

[Dil88] D. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. Technical Report 88-119, Carnegie Mellon University, Computer Science Dept, 1988.

[GS] S. M. German and A. P. Sistla. Reasoning about systems with many processes. GTE Laboratories Inc., Waltham, Massachusetts.

[Hen88] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[HK88] Z. Har'El and R. P. Kurshan. Software for analysis of coordination. In *Proc. Internat. Conf. Syst. Sci. Eng.*, pages 382–385. Pergamon, 1988.

[HK90] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. *AT&T Tech. J.*, 69:45–59, 1990.

[KK86] J. Katzenelson and R. P. Kurshan. S/R: A language for specifying protocols and other coordinating processes. In *Proc. 5th Ann. Int'l Phoenix Conf. Comput. Commun.*, pages 286–292. IEEE, 1986.

[Kur85] R. P. Kurshan. Modelling concurrent processes. In *Symp. Applied Math.*, volume 31, pages 45–57. Amer. Math. Soc., 1985.

[Kur87] R. P. Kurshan. Reducibility in analysis of coordination. In *LNCIS*, volume 103, pages 19–39. Springer-Verlag, 1987.

[Kur90] R. P. Kurshan. Analysis of discrete event coordination. In *Lecture Notes in Computer Science*, volume 430, pages 414–453. Springer-Verlag, 1990.

[McM92] K. L. McMillan. *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, May 1992.

[Mil80] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[SG89] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In *Workshop on Automatic Verification Methods for Finite State Systems*, LNCS, pages 151–165. Springer-Verlag, 1989.

[Wal88] D. Walker. Bisimulations and divergence in ccs. In *Third Annual Symposium on Logic in Computer Science*, pages 186–192. Computer Society Press, 1988.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Workshop on Automatic Verification Methods for Finite State Systems*, LNCS, pages 68–80. Springer-Verlag, 1989.