A Structural Induction Theorem for Processes

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

K. McMillan
Carnegie Mellon University
Pittsburgh, PA 15213

1 Introduction

In verifying finite state systems such as communication protocols or hardware controllers, we may be required to reason about systems comprised of a finite but effectively unbounded number of components. Examples are a network with an unspecified number of hosts, a multiprocessor with an unspecified number of CPU's, or a queue with an unspecified number of buffers. We would like to show that the system performs a certain set of tasks, regardless of the number of components. There are two problems which prevent the direct application of automatic verification systems which use state-space search (e.g., COSPAN [HK88]) to such a problem. The first problem is that such methods can be applied directly only to a fixed state space; it is generally not possible to quantify over the number of processes. The second problem is commonly referred to as the state space explosion problem. In principle, the verification method could be applied exhaustively to the 1-process system, the 2-process system, etc., until the largest possible system was verified. In practice, the fact that the number of states in a system increases geometrically with the number of components makes this approach infeasible. We present an induction method that allows us to infer properties of systems of unbounded size, but constructed by a uniform rule, from properties automatically verified on a system of fixed (and, presumably, small) size. The basis of this method is the ***structural induction theorem*** for processes.

Three methods have been described previously for verifying properties of systems with an unbounded number of identical processes. Homomorphic reduction [Kur85, Kur87] 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.

The method presented here is not based upon temporal logic, but instead makes use of a partial ordering on processes. This partial order might be, for example, the language containment relation, as in [Kur85, Kur87] or any of the various pre-orders defined for CCS processes[Hen88, Wal88]. A "specification", in the sense used here, is itself a process. Hence this method also has the advantage that a verified specification of a system may be used as an abstraction of that system for reasoning in a larger context. Also, there is no requirement that all systems, regardless of size, 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 partial order. For example, if we use the language containment order 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 partial order, this invariant process must be less than (or equal to) the specification and must be greater 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 greater in the partial order 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 less than (or equal to) 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. If the above three checks are carried out automatically, it is easy to experiment with different invariants until a correct one is found.

The following sections will add precision to the above discussion, 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 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 its full generality and proved. The general theorem includes the notion of a "shift operator" on processes, which allows us to induct on systems of processes with various communication topologies. In section 4, 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[HK88] to prove the correctness of the algorithm automatically.

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. 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.

We can formalize the above algorithm in a straightforward way using Milner's Calculus of Communicating Systems (CCS). In order to simplify matters, we will assume that the data object is a single bit 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 are also 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}). 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; [1] or it may 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. As the owner, a process may send read messages, or it may send out a copy message, in which case n is incremented by one. The owner may also send out an invalidate message, in which case n is decremented by one. If $n = 0$, the processor is the sole owner 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. The CCS description of this algorithm is then as follows:

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

We will consider the externally visible behaviors of our distributed object with respect to the simple process *Bit*, which we define below:

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

What we are going to prove is that the parallel composition of an arbitrary number of processors has no visible behaviors which are not behaviors of *Bit*. We assume that in the initial state, there is only one copy of the object in the system, and that the object's value is $v = 0$. Our goal, then, is 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

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

Again by monotonicity, we infer from 2 and 4 that

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

and hence,

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

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, of course, the process Q, which we will call a process invariant. The crucial property of Q is that it is greater in the preorder than its composition with Iv. We know of no direct method for discovering a process invariant, however we can present some heuristics. The invariant process which we discovered for the proof at hand is as follows:

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

The first thing to notice about Q is that it has precisely the same structure as $Ow(0,0)$, except that we have added a few transitions to a new state T. T represents the "top" process in the "may" preorder, that is it may accept any observable behavior on the set of actions we have defined. It should be noted that if we restrict the "invisible" actions α_{cp}, α_{ot} and α_{inv}, it is not possible for Q to reach the state T. In fact, $Q \setminus \{\alpha_{cp}, \alpha_{ot}, \alpha_{inv}\}$ is observationally equivalent to $Bit(0)$. If we think of Q as representing some arbitrary subset of the processors in our system, then its definition begins to make

[1] Presumably, such a message is preceded by some form of request message from the receiver of the data, but from our point of view, the request/acknowledge pair is considered atomic.

some sense. 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 a α_{cp} message, since it contains a process in the invalid state. If $Q \parallel Iv \leq_{may} Q$, then Q must also accept this message. However, since $Q = Ow'(0,0)$ represents a state where the owner of the object is inside our subset, we know this message will never be sent from outside the subset. 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 draw map ρ from the reachable states of Q to the reachable states of $Q \parallel Iv$. (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 $\rho^{-1}(A) \xrightarrow{\alpha} \rho^{-1}(B)$ or $\rho^{-1}(A) \xrightarrow{\alpha} T$. Our definition of ρ is as follows:

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

We complete our definition of Q by the following procedure. 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 designed to be "correct by construction". Instead, we used the Concurrency Workbench [2] to prove automatically that our definition of Q satisfies (1) and (2), and (3). Thus, using an automatic verifier and some simple inductive reasoning, we have been 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 very general induction theorem.

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

We begin with a class of "processes" $\mathcal{P}$. On this domain, we define a binary operator $\otimes$, a unary operator ϕ, and a partial order $\leq$. The operator $\otimes$ produces the (parallel) composition of two processes. A system composed of an array of processes $p_1, p_2, ..., p_n$ would be represented $p_1 \otimes p_2 \otimes \ldots \otimes p_n$, which we will 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 ϕ is called the shift operator; ϕ maps the process p_i onto process p_{i+1}, and in its simplest form is a renaming of variables. 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 indistinguishable. In rough analogy to a traditional induction proof, the shift operator represents the successor function on the inductively defined basis set. The partial order in the previous example was the "may" preorder on CCS processes. The literature contains a number of other examples of such orders, including the language containment relation for s/r-processes, the strong or weak bisimulation pre-order for CCS processes, and the "implements" relation for I/O automata.

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

- The composition operator $\otimes$ must be monotonic with respect to $\leq$, that is
$$\forall p,q,r \in \mathcal{P} : p \leq q \Rightarrow p \otimes r \leq q \otimes r \qquad (5)$$
- The shift operator ϕ must distribute over $\otimes$, i.e.
$$\forall p,q \in \mathcal{P} : \phi(p \otimes q) = \phi(p) \otimes \phi(q) \qquad (6)$$
- The shift operator ϕ must preserve the relation $\leq$, i.e.,
$$\forall p,q \in \mathcal{P} : p \leq q \Rightarrow \phi(p) \leq \phi(q) \qquad (7)$$

The structural induction theorem can now be stated as follows:

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

if
$$\bigotimes_{i=1}^{m} p_i \leq q_m \qquad (8)$$
and
$$q_m \otimes p_{m+1} \leq q_{m+1} \qquad (9)$$
then
$$\bigotimes_{i=1}^{n} p_i \leq q_n \qquad (10)$$

Proof. Applying properties (7) and (6) of ϕ to proposition (9) above, we have

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

Thus, since $p_{i+1} = \phi(p_i)$ and $q_{i+1} = \phi(q_i)$,

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

and, by inductive application of the above two steps,

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

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 inductive application of the above two steps,

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

thus, in particular,

$$\bigotimes_{i=1}^{n} p_i \leq q_n \tag{17}$$

□

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 from $P \leq T$, 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 $q'_n \leq T$. (The latter is conditional on the restriction or projection preserving the partial order).

4 Applying the general theorem

In order to illustrate the various constructs in our induction framework, and the generality of theorem 1, we will apply the theorem to a process formalism which is different from CCS in many respects: the L-processes of the *s/r model*. [3] The example we will use is 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]. Although the reduction in the previous analysis was not automatically checked, the only substantial part of the current proof which is not automatically checked is the proof of the structural induction theorem itself. The purpose of this example is to show how "eventuality" or "liveness" properties may be proved using the method, and also to illustrate how the induction theorem may be applied to systems with a ring-like communication structure.

Since L-processes are not as familiar as the calculus of communicating systems, we will say a few words about the process formalism before discussing the algorithm. An L-process p is a matrix over a Boolean algebra L. Associated with the Boolean algebra L is a set of variables, and with each variable, a domain. These variables are the means by which processes communicate, and are the only outwardly observable aspect of their behavior. An atom in the Boolean algebra is an assignment to each variable of a value in its domain. An element of the Boolean algebra is a disjunction (or sum) of atoms.

We can think of p_{ij} (the element of L in row i and column j of the matrix p) as being the condition for p to make a transition from state i to state j. An infinite sequence $a_0, a_1, \ldots$ of atoms in the Boolean algebra L is in the language of p (denoted $\mathcal{L}(p)$) if and only if there is a sequence of states $s_0, s_1, \ldots$ such that $a_i \in p_{s_i s_{i+1}}$, and s_0 is the initial state. The parallel composition of two L-processes p_1 and p_2 is $p_1 \otimes p_2$, their tensor product over the conjunction (product) operator of the Boolean algebra L. This definition has the property that a sequence of atoms is in $\mathcal{L}(p_1 \otimes p_2)$ if and only if it is in $\mathcal{L}(p_1)$ and in $\mathcal{L}(p_2)$, *i.e.*, $\mathcal{L}(p_1 \otimes p_2) = \mathcal{L}(p_1) \bigcap \mathcal{L}(p_2)$.

In order to express eventuality properties of L-processes, we introduce the notion of infinitary "exception" conditions. We denote by $\mathcal{C}(p)$ a set of subsets of the states of p called the *cycle sets* of p. In an L-process with cycle sets, a given sequence of atoms $a_0, a_1, \ldots$ is in the language of p *iff* there is a sequence $s_0, s_1, \ldots$ of states beginning with the initial state, such that $a_i \in p_{s_i s_{i+1}}$ and such that the set of states occurring infinitely often in the sequence is *not* contained in any cycle set. In the composition of two L-processes with cycle sets, the cycle sets $\mathcal{C}(p_1 \otimes p_2)$ are formed by "lifting" the cycle sets $\mathcal{C}(p_1)$ and $\mathcal{C}(p_2)$ to the corresponding sets in the state space of the composition, namely the cartesian product of the component state spaces.

[3] L-processes are the semantic objects on which the *s/r* model of coordinating systems is based.

We use the language containment order as our partial order in this model. That is, we define $p_1 \leq p_2$ if and only if $\mathcal{L}(p_1) \subseteq \mathcal{L}(p_2)$. The parallel composition operator is thus easily shown to be monotonic in $\leq$, satisfying condition 8. Our shift operator ϕ will be defined as the mapping on L-processes induced by a homomorphism ϕ' on the Boolean algebra L. In our example, this Boolean algebra homomorphism is in turn induced by an renaming ϕ'' on the variables of L.

Since one of the properties of a Boolean algebra homomorphism is that it distributes over the Boolean product operator, condition 9 is satisfied. Finally, although we do not prove it here, one of the major theorems of L-process theory states that Boolean algebra homomorphisms preserve the language containment relation, thus condition 10 is satisfied.

We assume the reader is familiar with Dijkstra's (in)famous Dining Philosopher's paradigm. Our algorithm for preventing deadlock, and hence starvation of the philosophers, breaks the symmetry of the problem by introducing a token (an encyclopedia), which is passed around the ring of philosophers. 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 is an L-process $P = \bigotimes_{i=1}^{n} p_i$, where p_i is defined in figure 4. 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. The Boolean algebra L has n variables, $x_1 \dots x_n$, which all have the domain $\{think, hungry, eat, read\}$. The variable x_i is called the *selection* of p_i. In the definition of p_i,

$$\text{left}(i) = \begin{cases} i-1, & i > 1; \\ n, & \text{otherwise}, \end{cases}$$

$$\text{right}(i) = \begin{cases} i+1, & i < n; \\ 1, & \text{otherwise} \end{cases}$$

and the condition B_i, indicating "blocking" of process i, is defined as follows:

$$\begin{aligned} B_i \equiv\ & (x_{\text{right}}(i) = eat) + (x_{\text{left}}(i) = hungry) \\ & +(x_{\text{left}}(i) = eat) \end{aligned}$$

Each process p_i has a single cycle set, EAT, which represents an assumption that no philosopher will remain in the eating state forever (a convention of the philosopher's guild). What we are going to prove is that the first philosopher p_1 does not starve, regardless of n. Stated formally, this means that $\mathcal{L}(P) \subseteq \mathcal{L}(T)$, where T is defined in figure 4. The language of T is all sequences except those in which the selection x_1 is *think* or *eat* a finite number of times.

Now comes the hard part: discovering a shift operator ϕ and a process invariant q_m which will satisfy the conditions of theorem 1. Let's jump ahead for a moment and look at the solution, then we'll come back and discuss how the solution was arrived at. The variable renaming ϕ'' which induces the shift operator on processes is defined as follows. If $1 < i < n$, then $\phi''(x_i) = x_{i+1}$. Otherwise $\phi''(x_i) = x_i$. In other words, the first and last selection variables are mapped onto themselves, while all of the other variables are "shifted up" by one. The process invariant q_i, is defined in figure 4. Using the COSPAN [4] system [HK88], we are able to check automatically that

$$\mathcal{L}(p_1 \otimes p_2) \subseteq \mathcal{L}(q_2) \tag{18}$$

$$\mathcal{L}(q_2 \otimes p_3) \subseteq \mathcal{L}(q_3) \tag{19}$$

(where $q_3 = \phi(q_2)$. Since $p_{i+1} = \phi(p_i)$ and $q_{i+1} = \phi(q_i)$, for all $2 < i < n-1$ (the reader should verify this), theorem 1 allows us to infer that

$$\mathcal{L}(\bigotimes_{i=1}^{n-1} p_i) \subseteq \mathcal{L}(q_{n-1}) \tag{20}$$

(provided $n \geq 3$). The next step in the proof is the trick that allows us to prove properties about ring-structured systems. We check automatically, using COSPAN, that

$$\mathcal{L}(q_{n-1} \otimes p_n) \subseteq \mathcal{L}(T) \tag{21}$$

Using (20) and the monotonicity of $\otimes$, we substitute $\bigotimes_{i=1}^{n-1} p_i$ for q_{n-1} in (21) to conclude that

$$\mathcal{L}(\bigotimes_{i=1}^{n} p_i) \subseteq \mathcal{L}(T) \qquad \square$$

Provided we have faith in the lemmas checked by COSPAN, the above constitutes sufficient proof of the result we were after. It does not, however, provide much insight into *why* the result is true. To understand our choice of q_i and ϕ, the reader should think of q_i (where $2 \leq i < n-1$) as an abstraction of the first i processes, much in the way we thought of Q as an abstraction of an arbitrary subset of processes in the first example. State 0 represents the case when one of the first $i-1$ processes holds the encyclopedia. State 1 indicates that

[4]COSPAN is a proprietary software system which may be licensed from AT&T by universities for educational purposes. The input text for the example is available upon request.

$$p_i = \begin{array}{c} \\ \textbf{THINK} \\ \\ \textbf{EAT} \\ \\ \textbf{READ} \end{array} \begin{array}{ccc} \textbf{THINK} & \textbf{EAT} & \textbf{READ} \\ \left(\begin{array}{c} (x_i = \mathit{think})* \\ (x_{\text{left}}(i) \neq \mathit{read}) \\ (x_i = \mathit{hungry}) * B_i \end{array}\right. & (x_i = \mathit{hungry}) * \neg B_i & \left.\begin{array}{c} (x_i = \mathit{think})* \\ (x_{\text{left}}(i) = \mathit{read}) \end{array}\right) \\ \begin{array}{c} (x_i = \mathit{think})* \\ (x_{\text{left}}(i) \neq \mathit{read}) \end{array} & (x_i = \mathit{eat}) & \begin{array}{c} (x_i = \mathit{think})* \\ (x_{\text{left}}(i) = \mathit{read}) \end{array} \\ \begin{array}{c} (x_i = \mathit{read})* \\ (x_{\text{right}}(i) = \mathit{think}) \end{array} & 0 & \begin{array}{c} (x_i = \mathit{read})* \\ (x_{\text{right}}(i) \neq \mathit{think}) \end{array} \end{array}$$

$$\mathcal{C}(p_i) = \{\{EAT\}\}$$

Figure 1: The philosopher process, p_i.

$$T = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & (x_1 = \mathit{eat}) + (x_1 = \mathit{think}) & \neg((x_1 = \mathit{eat}) + (x_1 = \mathit{think})) \\ 1 & (x_1 = \mathit{eat}) + (x_1 = \mathit{think}) & \neg((x_1 = \mathit{eat}) + (x_1 = \mathit{think})) \end{array}$$

$$\mathcal{C}(T) = \{\{1\}\}$$

Figure 2: The task T

$$q_i = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & (x_i \neq \mathit{read}) & \begin{array}{c} (x_i = \mathit{read})* \\ (x_{i+1} \neq \mathit{think}) \end{array} & \begin{array}{c} (x_i = \mathit{read})* \\ (x_{i+1} = \mathit{think}) \end{array} & 0 \\ 1 & 0 & \begin{array}{c} (x_i = \mathit{read})* \\ (x_{i+1} \neq \mathit{think}) \end{array} & \begin{array}{c} (x_i = \mathit{read})* \\ (x_{i+1} = \mathit{think}) \end{array} & 0 \\ 2 & \begin{array}{c} (x_i \neq \mathit{read})* \\ (x_n = \mathit{read})* \\ (x_1 = \mathit{think}) \end{array} & 0 & \begin{array}{c} (x_i \neq \mathit{read})* \\ ((x_1 = \mathit{eat}) + (x_n \neq \mathit{read})) \end{array} & \begin{array}{c} (x_i \neq \mathit{read})* \\ (x_n = \mathit{read})* \\ (x_1 = \mathit{hungry}) \end{array} \\ 3 & \begin{array}{c} (x_i \neq \mathit{read})* \\ (x_n = \mathit{read})* \\ (x_1 = \mathit{think}) \end{array} & 0 & \begin{array}{c} (x_i \neq \mathit{read})* \\ ((x_1 = \mathit{eat}) + (x_n \neq \mathit{read})) \end{array} & \begin{array}{c} (x_i \neq \mathit{read})* \\ (x_n = \mathit{read})* \\ (x_1 = \mathit{hungry}) \end{array} \end{array}$$

$$\mathcal{C}(q_i) = \{\{0\}, \{3\}\}$$

Figure 3: The process invariant q_i

process i, the last in our subset, holds the encyclopedia. State 0 forms a singleton cycle set, to enforce the eventuality that the encyclopedia reaches p_i. States 2 and 3 represent the case when the encyclopedia is outside our subset. These two states are identical, except that state 3 is entered whenever philosopher 1 refuses to accept the encyclopedia from philosopher n, i.e., when $x_n = read$ and $x_1 \neq think$. State 3 is also a singleton cycle set, to indicate that philosopher 1 must eventually accept the encyclopedia. The shift operator works as follows. To go from q_i to q_{i+1}, x_1 and x_n map onto themselves, but x_i and x_{i+1} are "shifted up" by one. This scheme works for all of the cases up to q_{n-1}, but breaks down going from q_{n-1} to q_n, since x_n is not shifted up. To get around this problem, we only carry the induction up to $n-1$, and then insert the process p_n at the end in order to "close the loop". This last step is somewhat analogous to the step in the first example, where we proved $Q \setminus \{\alpha_{cp}, ot, inv\} \leq_{may} Bit(0)$.

As in the first example, we can draw a map ρ from the states of the process invariant, to the states of the invariant composed with the "next" process, i.e., from q_{i+1} to $q_i \otimes p_{i+1}$. Our definition of ρ is as follows:

$$\begin{aligned}
\rho(0) &= \{(0,\mathbf{EAT}),(0,\mathbf{THINK}),\\
&\qquad (1,\mathbf{EAT}),(1,\mathbf{THINK})\}\\
\rho(1) &= \{(2,\mathbf{READ}),(3,\mathbf{READ})\}\\
\rho(2) &= \{(2,\mathbf{EAT}),(2,\mathbf{THINK})\}\\
\rho(3) &= \{(3,\mathbf{EAT}),(2,\mathbf{THINK})\}
\end{aligned}$$

The reader might want to verify that ρ is a homomorphism, that is, for all $i' = \rho^{-1}(i)$ and $j' = \rho^{-1}(j)$,

$$(q_m \otimes p_{m+1})_{ij} \leq (q_m + 1)_{i'j'}.$$

This homomorphism is used by COSPAN when checking proposition 19, to avoid the exponential state expansion involved in complementing q_3. In addition to checking that ρ is a valid homomorphism, it must also check that there are no infinite state sequences "excepted" by the cycle sets of q_{i+1} which map onto unexcepted sequences in $q_i \otimes p_{i+1}$. This can be done by verifying that if any strongly connected component in the state space of $q_i \otimes p_{i+1}$ is mapped into a cycle set of q_{i+1}, then it is also in a cycle set of $q_i \otimes p_{i+1}$.

Having verified that one processor in our dining philosophers algorithm cannot starve, we would like to hint at how one might verify that none of the philosophers starve. Of course, one uses another induction. The following is the approach we might take: First, define a new process invariant $q_{i,j}$, which is isomorphic to q_i, except that it represents a the set of process in the range $[i,j]$. Define a task T_k which is isomorphic to T, except that it specifies that philosopher p_k never **starves. We have already shown that**

$$\mathcal{L}(\bigotimes_{i=1}^{k-1} p_i) \subseteq \mathcal{L}(q_{1,k-1})$$

To carry out the second induction, we define new shift operator γ, such that for $1 < i < n$, $\gamma''(x_i) = x_{i-1}$, and otherwise $\gamma''(x_i) = x_i$. Using γ, we induct from $q_{n-1,n}$ to show that

$$\mathcal{L}(\bigotimes_{i=k}^{n} p_i) \subseteq \mathcal{L}(q_{k,n})$$

We then use COSPAN to check that

$$\mathcal{L}(q_{1,k-1} \otimes q_{k,n}) \subseteq \mathcal{L}(T_k)$$

By the monotonicity of $\otimes$, we conclude that

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

for any $2 < k < n$. (At this point, the reader should be able to guess how we check the cases $k=2$ and $k=n$).

We should note 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 $q_7 = \bigotimes_{i=1}^{7} p_i$. That is, the first seven processes work as a process invariant, although no number less than seven works. Why this is the case remains a mystery, but it does suggest a very simple approach to finding a process invariant which one might want to begin with, before attempting any deeper analysis.

5 Summary and Questions for Further Study

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. In our examples, we dealt we two different communication structures: an abstract, fully connected network, and a ring. It should be clear that the theorem is also general enough to apply to other structures, and combinations of these structures (for example, a parallel bus structure with a "daisy chained" arbitration system). 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 temporal logic be preserved between systems of varying numbers of processes.

As in the case of verification of program loops, the proof procedure requires the discovery of a kind of invariant: the process invariant q_i. The question then naturally arises of whether or not such an invariant process always exists, if a given systems satisfies a task, and the conditions of the theorem are met. This is, in some sense, a question of completeness of the method, and one which we have not yet addressed. Methods for deriving process invariants also warrant further study. Perhaps a certain style of design will be found to facilitate construction of process invariants, in much the same way that "well structured" programming facilitates the verification of sequential program.

6 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.

[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.

[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.

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

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