A Structural Linearization Principle for Processes

R. P. Kurshan[1] M. Merritt[1] A. Orda[2] S. R. Sachs[3]

[1] AT&T Bell Labs, Murray Hill, NJ 07974, USA
[2] Dept. of Elec. Eng., Technion, Haifa 32000, Israel
[3] Dept. of Elec. Eng. and CS, U. C. Berkeley, Berkeley, CA 94720, USA

Abstract. In [1], an induction principle for processes was given which allows one to apply model-checking techniques to parameterized families of processes. A limitation of the induction principle is that it does not apply to the case in which one process depends directly upon a parameterized number of processes, which number grows without bound. This would seem to preclude its application to families of N processes interconnected in a star topology. Nonetheless, we show that if the dependency can be computed incrementally, then the direct dependency upon the parameterized number of processes may be re-expressed recursively in terms of a linear cascade of processes, yielding in effect a "linearization" of the inter-process dependencies and allowing the induction principle to apply.

1 Introduction

Finite state systems such as communication networks, distributed multi-processor systems and hardware controllers, are many times specified as comprising of a finite but unbounded number of components. In verifying such a specification we aim at showing that it performs a certain task regardless the actual (finite) number of components. Nevertheless, automatic verification systems (e.g., Cospan [2]) cannot directly handle such verification problems, since they apply only to a fixed state space, and in general they do not permit a quantification on the number of processes. A simple solution might be to test the specification against any possible number of components, up to the largest possible (practical) number; nonetheless, the number of states in a system increases geometrically with the number of components, making this approach intractable.

A number of authors have dealt with this problem. For example, Browne, Clarke and Grumberg[3] introduced a method which uses an indexed form of branching-time temporal logic, applicable to families of processes in case the next-time operator is removed from the logic and process quantifiers are not nested (so global properties of the system cannot be stated). A method due to German and Sistla[4] applies to a linear-time temporal logic (again, without the next-time operator). Although they can verify global properties, their decision algorithm is doubly exponential in the process size. Shtadler and Grumberg[5] propose a method which requires the user to formulate a "network grammar"

by which processes may be recursively defined. They also exclude the next-time operator from their logic.

A solution approach to the above was presented in [1]: an Induction Principle for processes was given which allows one to apply model-checking techniques to parameterized families of processes. With the Induction Principle, one can infer properties of systems of unbounded size through an automatic verification carried on a system of fixed (and hopefully small) size. The Induction Principle is based on providing an "invariant" process that carries out the inductive step. Obviously, the invariant cannot depend on the actual size of the system. In effect, the invariant encapsulates that part of the system's logic that would enable a "hand-written" inductive proof. This verification process applies to any ω-regular property, (including global properties), and once an invariant is found, the verification step is linear in the size of the process and the invariant.

A limitation of the Induction Principle is that it does not apply to the case in which one process or one variable depends directly upon a parameterized number of processes, which grows without bound. This is the case, for example, with a star topology, in which a "hub" talks with each other component of the system; as the system grows, the hub has to deal with a larger number of components, meaning that its representation per se depends on the system size. Indeed, this would seem to preclude its application to families of N processes interconnected in a star topology. In fact, such a limitation may seem natural, as in a star topology, it is the "role" of the hub to accumulate information from the other N components, thus no invariant independent of this number may exist, as required for the Induction Principle to apply.

However, under certain circumstances there is a way out. In this paper we show that if the direct dependency of the hub (or of a bounded number of such processes) upon the parameterized number of processes can be computed incrementally, then this dependency may be re-expressed recursively in terms of a linear cascade of processes, yielding in effect a "linearization" of the inter-process dependencies (linear in the sense of a cascade or shift register), allowing the Induction Principle to apply. Throughout the paper we depict the problem and its solution through an example of a distributed "doorway" algorithm for a star-like topology [6].

2 Doorway Algorithm Example

The Linearization Principle is illustrated by a distributed algorithm whose interconnectivity has a star topology. In this section we present the algorithm and describe the problems encountered with its formal verification; then, we outline the solution approach to these problems, which is based on the Linearization Principle presented in this paper.

2.1 The Doorway Algorithm

In [6] several algorithms are presented for handling faulty shared memory. One of these is the Doorway algorithm described herein. The doorway algorithm consists

of test-and-set (t&s) operations performed by some N processes ($P^1 \cdots P^N$) on four t&s registers, one of which may be faulty. The reader is referred to [6] for a full description of a t&s register operation. For the purposes of this paper, the following concise description suffices.

A process P^i ($i = 1, \cdots, n$) interacts with a t&s register by presenting to it a t&s *request*. The process is guaranteed to get a response within finite time (even if the register is faulty). The response can be either *win* or *lose*. The first response of a nonfaulty register must be *win*, and all subsequent ones must be *lose*. A faulty register, on the other hand, may produce any sequence of *win-lose* responses. It is assumed that a register (either faulty or nonfaulty) gives at most one response at any given time, which corresponds and is forwarded to one of the processes that have presented a t&s request to it.

The goal of the Doorway algorithm is to filter either one or two processes out of the list of (at most) N processes that present t&s requests. These one/two processes are later forwarded to a second algorithm (not discussed in this paper), which chooses exactly one process as a final winner. As previously mentioned, the Doorway algorithm is based on a construction of four t&s registers, of which at most one may be faulty. For simplicity, it can also be assumed that, at any given time, at most one register produces a win/lose response (however, it is straightforward to adapt all that follows to the more general case where such an assumption is not made). The doorway consists of two "half-doorways" D_1 and D_2, each consisting of a pair of t&s registers.

The algorithm works as follows. A process that wants to pass through the doorway presents a t&s request to the two registers of the first half-doorway D_1. If the process gets a *win* response from at least one of these two registers, it moves on to the second half-doorway D_2; otherwise, the process declares itself as a "loser" of the doorway. A process moving to D_2 presents a t&s request to its two registers. If the process gets a *win* response from at least one of these two registers, it declares itself as a "winner" of the doorway; otherwise, the process declares itself as a "loser".

The task that the Doorway algorithm has to perform is stated as follows: if there is some t&s request from some process P^i ($i = 1, 2, \cdots, n$) then eventually at least one and at most two processes among those that have requested declare themselves as winners of the doorway, whereas all others that have requested declare themselves as losers of the doorway.

2.2 The Doorway Problem and its Solution

Given a specific value for the number of processes N, a formal verification tool such as Cospan [2] can be used in order to verify that the Doorway algorithm indeed performs its task (the reader is referred to [7] for a detailed discussion on verification issues of t&s algorithms using Cospan). The problem comes when one wants to verify this for *any* finite number N. One cannot hope to apply the Induction Principle of [1] directly, since the Doorway algorithm suffers from the deficiencies outlined in the previous section. This is depicted in the following.

Consider the following way of modeling the doorway scheme, depicted in Figure 1. The four registers of the doorway are modeled by two "half-doorway" processes D_1 and D_2, that produce "win" and "lose" signals. We have also N processes P^i $(i = 1, 2, \cdots, N)$, that may present t&s requests to each of the two half-doorways, according to the signals received from the half-doorways and to the Doorway algorithm. Between the half-doorways and the processes stands a "selector" S, that selects the addressees of the half-doorways responses; i.e., whenever a half-doorway sends a "win" or "lose" signal, the selector should choose one out of a list of processes that have presented a request to that half-doorway and have not been answered yet. The signal is forwarded to the chosen processor through a global variable X_{ij} i.e., the response of the j-th half-doorway to the i-th process. The problem with the above lies in its intrinsic dependence

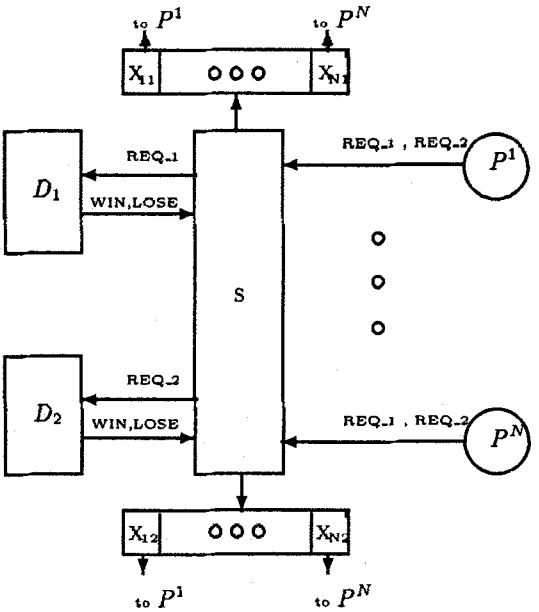

Fig. 1. Doorway as Star Topology

on the number N. For example, the selector S should communicate explicitly with each of $P^1, \cdots, P^N$, and thus it is a parametrical process that depends on the number N. Another example are the global variables X_{ij}, whose number depend also on N. We seek a representation of the system, in which there is no dependency on N, neither in terms of variables nor states nor transitions between states.

The solution for the doorway case is depicted in Figure 2, which describes a *linearization* of the initial star topology. The figure shows the de-parametrization

of the selector S. The selector is distributed among N bits S^i, each locally assigned to an "outer-layer" of P^i. Each S^i selects (at every slot) a binary value, which corresponds to P^i being or not being selected at that slot. In order to have a legal selection, we must make sure that exactly one process P^i is selected at any slot. This is accomplished by propagating the selections of the S^i's backwards, from S^N up to S^1: the propagation is performed by having a variable S_p^i at the outer level of P^i, whose value depends on that of S^i and that of S_p^{i+1}. The role of S_p^i is to report whether there were 0, 1 or more selections in the S^j bits in the range S^i-S^N. Thus, the legal cases are those for which $S_p^1 = 1$. We eliminate the forbidden cases through automaton acceptance conditions (the structures to which these apply must be linear, as well). The reason this construction works is that the propagated signal is of bounded range, and thus independent of N.

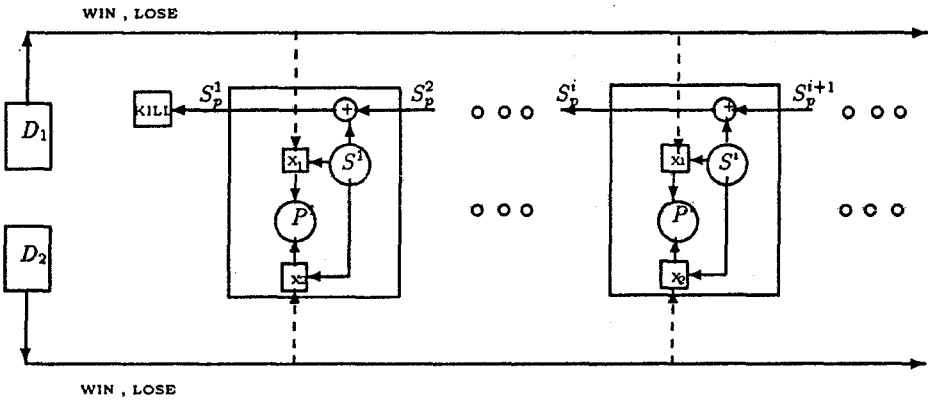

Fig. 2. Doorway as Linear Topology: selector and X var's

Figure 2 also shows how the global variables X_{ij} become local variables X_l^i of the outer-layers of the P^i's. In a similar way requests from all the processes can be accumulated along the chain and presented to the corresponding half-doorway through a bounded-range signal.

In the following sections we shall discuss in detail the linearization of the doorway scheme. The outcome of this linearization is a representation that can be accommodated within the Induction Principle. The above example hints at the properties of the doorway scheme that enabled its linearization e.g., the ability to represent the legal and nonlegal cases of a distributed selector within a bounded range. In the following section we formally discuss the Linearization Principle and present the criteria needed for it to apply.

3 Linearization Principle

Linearization is applied to a system of *processes* (defined in Section 3.1) $S(N) = \{P^1, \cdots, P^M\}$ (parameterized by N), when one or at most a bounded number of the processes have $O(M)$ dependencies upon the other processes, the remaining processes having $O(1)$ such dependencies. The effect of linearization is to re-implement the dependencies recursively, serving to "flatten" a star dependency topology into a "linear" topology (like a cascade or shift register).

A system of the form $S'(N) = \{P^1, \cdots, P^M; x_1, \cdots, x_K\}$ consisting of processes $P^1, \cdots, P^M$ and "system-level" variables $x_1, \cdots, x_K$, which inter-connect the processes, may be reduced to the form of $S(N)$ by letting $x_1, \cdots, x_K$ be variables of respective (1-state) processes $Q^1, \cdots, Q^K$ and setting

$$S(N) = \{P^1, \cdots, P^M, Q^1, \cdots, Q^K\}.$$

Linearization is based upon augmenting the given system with variables (or processes), whose values are defined in terms of a finite recursion. Those process dependencies upon $O(M)$ processes are replaced with respective dependencies upon $O(1)$ processes, including those added. When this is possible, we say $S(N)$ is *linearizable.*

We prove that a system of processes $S(N)$ is linearizable if the process dependencies can be computed incrementally.

3.1 Process Systems

The concept of finite-state "process" is common to many models of coordination. The Linearization Principle presented here applies to many of these models. We characterize process in terms of variables. This characterization is equivalent to the characterization in terms of Boolean algebra [8] used to support the development of homomorphic reduction. The characterization in terms of variables contains more details at an "implementational" level, as required for our development of the Linearization Principle. Transformation of the Linearization Principle to other process characterizations, such as those in terms of Boolean algebra [8], traces [9, 10, 11, 12] or synchronization primitives [13] should be straightforward (either by encoding these characterizations within this one, or by re-casting the linearization Principle directly within the chosen framework).

We allow all process constructs to be parameterized by an integer index. For the system of processes $S(N)$, N is the parameter. We will describe how to construct a process $\widehat{S}(N)$ which is a "parallel composition" of the system $S(N)$. Thus, $\widehat{S}(N)$ defines an infinite family of (finite state) processes $\widehat{S}(1), \widehat{S}(2), \cdots$. The motive for applying the Linearization Principle is to support computer-aided verification of $\widehat{S}(N)$ for all N, through application of the Induction Principle [1]. Verification always is with respect to a property or "specification" defined by some process. Verification of $\widehat{S}(N)$ with respect to the property or specification defined by the process $P(N)$ consists of the language-containment test

$$\mathcal{L}(\widehat{S}(N)) \subseteq \mathcal{L}(P(N)).$$

While for each N this is a finite test, and for sufficiently small N, the test may be performed on a computer through exhaustive search, to check this for all N requires an infinite test. The Induction Principle reduces this infinite test to a finite test, under certain conditions. The Linearization Principle given here in effect broadens those conditions, by providing a transformation from a broader class of processes to the class in which the Induction Principle applies.

The foregoing motivates the following definitions.

Definition: A *process* is a function

$$P : V \times D^k \rightarrow 2^V \times (2^D)^k$$

for some finite sets $V = V(P)$ (the process *states*) and D (the domain of the process *variables*, defined below), and integer k, subject to the following:

writing $P(x) = (P_0(x), ..., P_k(x))$ for $x = (x_0, ..., x_k)$, for all $i > 0$, either $P_i(x) = \{x_i\}$ or P_i is independent of x_j for all $j \geq i$.

($P_i(x)$ is *independent* of x_j if for each (fixed) value of $(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_k)$, P_i is a constant function of x_j; otherwise, we say $P_i(x)$ *depends* upon x_j.)

We say x_0 is the *state* of P, $P_0(x)$ is the *state transition relation* of P and $x_1, \cdots, x_k$ are the *(combinational) variables* of P. The *number of variables* of P is $k(P) = k$. The range of P is $2^V \times (2^D)^k$, rather than simply $V \times D^k$, in order to support non-deterministic state transitions and variable assignments. Often we will abuse notation and write a value in place of the singleton set containing that value. Thus, for example, we may write $P_i(x) = x_i$ to mean $P_i(x) = \{x_i\}$. When $P_i(x) = x_i$, x_i (or P_i) is called an *input* to P; otherwise x_i (P_i) is called an *output* of P. Let $\widehat{k}(P)$ denote the number of outputs of P. If $\widehat{k}(P) = k(P)$, P is said to be *closed*. We do not define the *language* $\mathcal{L}(P)$ of the process P, as that is not germane to our discussion of the Linearization Principle. However, in order to place the above definition in context, we note that customarily, $\mathcal{L}(P)$ is defined in terms of successions of states (values of x_0), each succession starting with a state in a given set of *initial* states $I \subseteq V$, the succession possibly required to satisfy some *termination, acceptance* (or "fairness") conditions, and such that for each pair of consecutive states v, w in the given succession, there exists some $x = (v, x_1, ..., x_k)$ with

$$w \in P_0(x)$$

and

$$x_i \in P_i(x)$$

for all $i > 0$. (This makes sense on account of the variable independence assumptions.) $\mathcal{L}(P)$ is the set of sequences or strings $\mathbf{t}$ of successive values t_i of the vector $(x_m, \ldots, x_k)$ for some fixed m, each such sequence relative to a succession of states as described. The variables $x_1, \ldots, x_{m-1}$ (which may be inputs or outputs) are *internal* variables, not represented in $\mathcal{L}(P)$.

Example 1: a) Define the process $P(x)$ with $k(P) = 3$ and $V(P) = D = \{0, \ldots, 2^{32} - 1\}$ for $x = (x_0, x_1, x_2, x_3)$ by

$$\begin{aligned} P_0(x) &= x_1 x_2 + (1 - x_1) x_0 \ , \\ P_1(x) &= \{0, 1\} \ , \\ P_2(x) &= x_2 \ , \\ P_3(x) &= x_0 \ . \end{aligned}$$

This process has a "control" variable x_1 which nondeterministically assumes the values 0 or 1. If $x_1 = 1$, P assigns the value of its input x_2 as its next state; otherwise its state remains unchanged. The process produces the value of its current state in its output x_3. The states v, w are consecutive if for $x_1 = 0$ or $x_1 = 1$ and some x_2,

$$w = P_0(v, x_1, x_2, v) = x_1 x_2 + (1 - x_1) v \ .$$

(Thus, every pair of states v, w are consecutive.) If x_1 is taken to be internal, then $\mathcal{L}(P)$ consists of successions of pairs (a_i, b_i) where $b_{i+1} = a_i$ or $b_{i+1} = b_i$.

b) Let $Q(x)$ be the process with $k(Q) = 3$ and $V(Q) = D$ as above, defined by

$$\begin{aligned} Q_0(x) &= x_0 + x_1 x_2 \ , \\ Q_1(x) &= x_1 \ , \\ Q_2(x) &= x_2 \ , \\ Q_3(x) &= x_0 \ . \end{aligned}$$

(Say, arithmetic is modulo 2^{32}.) This process takes the product of inputs x_1, x_2 and adds that to its current state x_0, whose value it produces in output x_3. The states v, w are consecutive if for some x_1, x_2

$$w = Q_0(v, x_1, x_2, v) = v + x_1 x_2 \ .$$

Thus, v, w are consecutive as long as $w \geq v$. If x_1 is taken to be internal, then $\mathcal{L}(Q)$ consists of successions of pairs (a_i, b_i) where a_i is arbitrary and $b_{i+1} = b_i + c_i a_i$ for some $c_i \geq 0$.

Definition: A *process system* is a finite set of processes $S = \{P^1, \ldots, P^M\}$ sharing a common set of states, $V(S)$, and a common variables domain, say D. The *variables* of S are the variables of the processes in S. A *parallel composition* of the system of processes S is a process $\widehat{S}$ with states $V(\widehat{S}) = V(S)^M$, variables domain D and $\Sigma_{i=1}^M \widehat{k}(P^i) \leq k(\widehat{S}) \leq \Sigma_{i=1}^M k(P^i)$ variables, subject to the following: for $k = k(\widehat{S})$ and $y = (v, y_1, \ldots, y_k)$, where $v = (v_1, \ldots, v_M) \in V(S)^M$, and for some $k(P^i)$-dimensional vector Y_i whose components are (not necessarily unique) elements of the set $y_1, \ldots, y_k$, $i = 1, \ldots, M$, it is required that for each $i > 0$,

$$\widehat{S}_i(y) = P_n^j(v_j, Y_j) \qquad (1)$$

for some $j, n > 0$ (*i.e.*, each variable of $\widehat{S}$ is a variable of S); furthermore, it is required that if $Y_{js} = y_m$ (*i.e.*, y_m is the s-th component of Y_j) then either P_s^j is an input to P^j ($P_s^j(x) = x_s$) or $\widehat{S}_m(y) = P_s^j(v_j, Y_j)$ (*i.e.*, y_m is in fact the s-th variable of P^j); finally, it is required that for each output P_n^j of S there is a unique i for which (1) holds, and the state transition relation $\widehat{S}_0$ of $\widehat{S}$ satisfies

$$\widehat{S}_0(y) = P_0^1(v_1, Y_1) \times \cdots \times P_0^M(v_M, Y_M) \tag{2}$$

(where $\times$ denotes Cartesian product), provided

$$y_i \in \widehat{S}_i(y) \tag{3}$$

for all $i > 0$. If (1) holds, and for some m, s, $Y_{ms} = y_i$, then, for $s < t$, we say P_t^m *depends* upon P_n^j if $P_t^m(x)$ depends upon x_s.

Thus, the variables of $\widehat{S}$ are the variables of S (or, more precisely, each component $\widehat{S}_i$ of $\widehat{S}$ ($i > 0$) is equal to a (non-state) component of some P^j, evaluated on a subset of the arguments of $\widehat{S}$). Since $\widehat{S}$ is a process, these variables must satisfy the independence assumption for processes. The association (1) defines the interconnectivity of the processes in S: if y_i is a component of Y_m and $m \neq j$, then P^m takes as "input" the n-th variable of P^j. In this case, the output y_i of the parallel composition $\widehat{S}$ is internal *iff* both P_n^j and the corresponding input of P^m are internal. (Unsubstituted inputs retain their character– internal or not– in $\widehat{S}$.)

Example 2: Let $S = \{P, Q\}$, where P and Q are taken from example 1. We may define a parallel composition $\widehat{S}$ of P and Q as follows. In this composition we define a closed process (no inputs) in which the following output-input match is made:

$\widehat{S}$	output	$\rightarrow$	input
y_1 :	P_1	$\rightarrow$	Q_1
y_2 :	P_3	$\rightarrow$	Q_2
y_3 :	Q_3	$\rightarrow$	P_2

Thus, for $y = ((v, w), y_1, y_2, y_3)$,

$$y_0 = \widehat{S}_0(y) = (P_0(v, y_1, y_3, \cdot), Q_0(w, y_1, y_2, \cdot)) = (y_1 y_3 + (1 - y_1)v, w + y_1 y_2)$$
$$y_1 = \widehat{S}_1(y) = P_1(v, y_1, y_3, \cdot) = \{0, 1\}$$
$$y_2 = \widehat{S}_2(y) = P_3(v, y_1, y_3, \cdot) = v$$
$$y_3 = \widehat{S}_3(y) = Q_3(w, y_1, y_2, \cdot) = w.$$

Here, "$\cdot$" indicates that the choice of variable is arbitrary. Since $y_1 = P_1$ is internal, $\mathcal{L}(\widehat{S})$ consists of successions of pairs (a_i, b_i) of values of (y_2, y_3), where $a_{i+1} = b_i$ and $b_{i+1} = a_i + b_i$ or $a_{i+1} = a_i$ and $b_{i+1} = b_i$. If each succession is initialized at $v = 1$, $w = 1$, then (until b_i first surpasses 2^{32}) $\mathcal{L}(\widehat{S})$ consists of Fibonacci pairs "computed asynchronously" (*i.e.*, with arbitrary repetitions).

The concept of "parallel composition" described above supports both the "synchronous product" illustrated in example 2, and "interleaving": the parallel

composition is *interleaving* provided for each component in (2), $P_0^i(v_i, \cdots) = v_i$ for all but at most one value of i, when (3) is satisfied. (See [14] for details.)

We assume that for each system of processes, some (fixed) parallel composition is defined (although which specific one, is not of interest here). Under suitable termination or acceptance semantics, a parallel composition with no internal variables satisfies the Language Intersection Property: $\mathcal{L}(\widehat{S}) = \mathcal{L}(P^1) \cap \cdots \cap \mathcal{L}(P^M)$. The Language Intersection Property, in turn, gives a convenient means for reorganizing process systems: if $S = \{P^1, ..., P^m, ..., P^M\}$ is a system of processes, then so are $T = \{P^1, ..., P^m\}$ and $U = \{\widehat{T}, P^{m+1}, ..., P^M\}$, and under consistent substitution of outputs for inputs (1), $\mathcal{L}(\widehat{S}) = \mathcal{L}(\widehat{U})$. Through such a reorganization, one can directly generalize those assertions of the Linearization Principle which apply to one process (see below), to several processes.

3.2 Linearization

Definition: A system of processes $T(N) = \{Q^1, \ldots, Q^M\}$ is *linear* if $k(Q^i) = O(1)$ for all i. Say $T(N)$ is a *linearization* of a parameterized system of processes $S(N) = \{P^1, ..., P^M\}$ if $T(N)$ is linear and $\mathcal{L}(\widehat{S}(N)) = \mathcal{L}(\widehat{T}(N))$. Say $S(N)$ is *linearizable* if it admits of a linearization.

Definition: A function $f : D^n \to D$ is *associative* if there exist $f_i : D^2 \to D$ $(i = 1, \ldots, n-1)$ such that for $z_i = f_i(x_i, z_{i+1})$ $(z_n = x_n)$, $f(x) = z_1$.

Definition: A variable x of a parameterized process is *of bounded computation* if the number of variables upon which x depends, is bounded.

Theorem 1. *A system $S(N) = \{P^1, \ldots, p^M\}$ is linearizable if*

(a) $k(P^i) = O(1)$ for all $i > 1$;
(b) $k(P^1) = O(M)$;
(c) every variable and the state of P^1 either is of bounded computation, or is associative;
(d) the number of variables of P^1 not of bounded computation, is bounded.

Proof: (sketch) First, we add an internal variable to P^1 which is assigned the value of the state P_0^1, to enable variables which depend upon it to be moved out of P^1. We distribute among $P^2, \cdots, P^M$ all variables of P^1 which are of bounded computation. This leaves $O(1)$ variables remaining in P^1. Since there are only $O(M)$ variables in the system, each of the remaining variables of P^1 can depend upon at most $O(M)$ variables. Therefore, for each of these $O(1)$ remaining variables x, we can use associativity to introduce an additional $O(M)$ internal variables z_i, $i = 1, \cdots, M$, associating each z_i with P^i, and then use the recursion guaranteed by associativity and z_1 to assign x. The resulting x has the same value as before, and is of bounded computation. Repeating this construction for each such x gives the required result. □

3.3 Mechanical Implementation

There is a potential for algorithmic linearization of a process system which meets the conditions of the above theorem. Processes whose number of variables is parameterized on N may be identified syntactically. Thus, it may be checked syntactically if the specified system meets the required conditions. If so, the construction of the theorem may be followed to give a linearization.

While most functions $D^n \to D$ are not associative (of the $|D|^{|D|^n}$ such functions, only $|D|^{n|D|^2}$ are associative), probably most if not all functions which would arise as parameterized variables of a a process in an engineering setting would be associative. To recognize these algorithmically would probably require that the user adhere to a pre-defined syntax when defining such parameterized variables. However, this may be a very natural requirement.

It may be convenient syntactically to identify the reassigned variables with respective "enveloping" processes, for each of the $O(M)$ system processes. Likewise, we may identify syntactically components of expressions which depend upon $O(M)$ variables, and replace each of these by new respective variables corresponding to the constructed recursion.

In order to verify that variables which are not of bounded computation (easily identified syntactically, through the presence of the parameter N), it may be necessary to use an interface to a theorem-prover (such as the interface described in [15]).

4 Application to the Doorway Example

4.1 Linearization of the doorway

In a previous section we outlined the way in which the doorway system is linearized. We are now ready to complete this description; due to length constraints, we shall describe only the "core" of the system. We note that we are implicitly addressing a specification in s/r code[2], which corresponds to finite state synchronous systems.

The half-doorways D_l $(l = 1, 2)$ are simple state-machines that produce win/lose signals according to the Doorway Algorithm. When a half-doorway is faulty, it may produce any sequence of such signals. Each half-doorway receives as input a corresponding signal R_l^1, which is passed to it from the first "envelope" process EP^1 (see below). This signal indicates to D_l whether some process P^i, $i = 1, \cdots N$, is requesting from D_l.

For each process P^i we define an "envelope" process EP^i. The envelope EP^i contains the following entities:

- The basic process P^i, which presents requests to each of the two half-doorways, gets win/lose responses through the X-variables (see below), according to the Doorway Algorithm.
- Local "win/lose" variables X_l^i $(l = 1, 2)$, indicating whether P^i has won or lost in the half-doorway D_l.

- A local "selector" process S^i with a binary selection, which decides whether EP^i is choosing P^i to be active at a given slot of time.
- A ternary "propagation variable" S_p^i, whose value describes whether there are 0, 1 or more processes in the range $[1..N]$ choosing themselves (i.e., whose local selector is set).
- A binary variable K^i which is set when any of the processes EP^j in the range $[i..N]$ selects itself and receives a response win or lose from a half-doorway D_l $(l = 1, 2)$, even though it didn't make a request to the responding half-doorway, resulting in the erroneous event that one P^j in the range $[i..N]$ might win or lose even though it is not requesting.
- "Request propagation" variables R_l^i $(l = 1, 2)$, indicating whether a process in the range $[i..N]$ is requesting from the half-doorway D_l.

Note that the value of a "propagation" variable (S_p^i, K^i, R_l^i) depends on that of a local variable and on that of another propagation variable (e.g., S_p^i is determined by S^i andS_p^{i+1}).

In order to finally establish our linearized doorway system, we have to get rid of some bad events caused by the distributed fashion in which selections are made. This is done by the incorporation of "Kill" processes, which can be thought of as additional processes that interact with $EP[1]$ and avoid the bad selections by looking at the values of its variables.

4.2 Formal Specification and Verification of the Linear Doorway

As previously mentioned, Cospan[2] was chosen as the framework for the specification and verification of the linearized doorway.

The modeling of the linear doorway in Cospan consists in specifying a system of processes. Each process P consists of a *declarations* and a *body*. In the former, the finite sets $V(P)$ (process states) and D (variables domain) are defined as well as the initial states of P. Also, undesirable process behavior can be removed by defining sets of states which the process must eventually leave (acceptance conditions). In the *body*, the actions of the process are captured by a transition structure with a (possibly) non-deterministic set of variables associated with every state.

As mentioned previously, a faulty half-doorway produces arbitrary responses, which in Cospan is modeled via the "free" construct. A "freed" process arbitrarily produces any of the variables defined for its domain D. Thus, either half-doorway may be specified to be "free" according to run time parameters.

The tasks that we require the linear doorway system to perform were described in a previous section. Their modeling in Cospan also consists of specifying them as processes. The only difference between a process which is a component of a system of processes and a process which specifies a task lies on the interpretation of the acceptance conditions.

Having specified the doorway system $S = \{D_1, D_2, P^1, P^2, \cdots P^N\}$ and each of the tasks T^j, Cospan calculates their *parallel composition* $\hat{S}(N)$ as defined in

section 3.1, and proceeds to test language containment for each task independently.

We proceed by stating the Induction Principle[1]. Let $\leq$ be a partial order on processes (e.g., one induced by a language containment order); let $\otimes$ be a composition operator on processes, monotonic with respect to $\leq$; let ϕ be a unary *shift* operator on processes, which distributes over $\otimes$ and preserves the relation $\leq$. We then have:

Induction Principle: Given two sets of processes, $\{p^1, p^2, \cdots, p^N\}$ and $\{q^1, q^2, \cdots, q^N\}$, 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

$$\otimes_{i=1}^{m} p^i \leq q^m \tag{4}$$

and

$$q_m \otimes p^{m+1} \leq q^{m+1} \tag{5}$$

then

$$\otimes_{i=1}^{N} p^i \leq q^N$$

Thus, by proving two propositions about arrays of fixed size m, we may draw a conclusion about an array of arbitrary size N. In particular, $q^N \leq T$ implies $\otimes_{i=1}^{N} p^i \leq T$, that is: in order to verify that the system $\otimes_{i=1}^{N} p^i$ performs task T it is sufficient to verify that q^N performs T. q^i is called the process *invariant*.

We now apply the Induction Principle to the doorway example. Denote $p^0 = D_1 \otimes D_2$, $p^i = EP[i-1]$ for $1 \leq i \leq N$. We choose the following invariant:

$$q^i = p^0 \otimes p^i \otimes p^{i-1} \otimes p^{i-2} \otimes p^{i-3} \hat{p}^{i-4}$$

where $\hat{p}$ is a modified version of p, whose variables are *freed* except for some of its "propagation" variables which are not allowed to take values which causes task T to fail. The shift operation on the processes p^i is defined as:

- for $i = 0$: $\phi(p^i) = p^i$;
- for $0 < i < N$: $\phi(p^i) = p^{i+1}$;
- for $\forall i$: $\phi(\hat{p}^i) = \hat{p}^{i+1}$;

i.e., ϕ is an identity map for the doorway and maps all other processes to their successors.

Choosing $m = 4$ we have that the "induction base" (4)

$$p^0 \otimes_{i=4}^{1} p^i \leq p^0 \otimes_{i=4}^{1} p^i \otimes \hat{p}^0$$

holds trivially because $\hat{p}^0$ allows q^i to have more behaviors than $p^0 \otimes_{i=4}^{1} p^i$. The "inductive step" (5) which is verified with Cospan is:

$$p^0 \otimes_{i=4}^{1} p^i \otimes \hat{p}^0 \otimes p^5 \leq p^0 \otimes_{i=5}^{2} p^i \otimes \hat{p}^1$$

Thus, by verifying that $q^N \leq T$ we have that $p^0 \otimes_{i=N}^{1} p^i \leq T$.

We note that a linearizable star topology presents a very simple case for the Induction Principle. The reason for this property is that the processes P^i's (for $i > 0$), all connected to the hub, do not talk with each other, making the ϕ-shifts be in fact "out of the picture". This indeed is not the case in other applications of the Induction Principle (e.g., the Dining Philosophers' problem discussed in [1]), for which finding proper ϕ's and q^i's may prove to be a hard task.

References

1. R. P. Kurshan and K. McMillan, "A structural induction theorem for processes," in *Proceedings of 8th ACM Symp. on Principles of Distributed Computing*, pp. 239–247, 1989.
2. Z. Har'El and R. P. Kurshan, "Modelling concurrent processes," in *Proceedings of Internat. Conf. Syst. Sci. Eng.*, pp. 382–385, 1988.
3. M. C. Browne, E. M. Clarke, and O. Grumberg, "Reasoning about networks with many identical finite state processes," in *In ACM Symp. Principles of Distributed Computing 5*, 1986.
4. S. M. German and A. P. Sistla, "Reasoning about systems with many processes," *GTE Laboratories Inc., Waltham, Massachusetts*, 1988.
5. Z. Shtadler and O. Grumberg, "Network grammars, communication behaviors and automatic verification," *LNCS*, vol. 407, pp. 151–165, 1989.
6. Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld, "Computing with faulty shared memory," in *Proceedings of 11th ACM Symp. on Principles of Distributed Computing*, 1992.
7. R. P. Kurshan, M. Merritt, A. Orda, and S. R. Sachs, "Formal verification of a distributed algorithm for accessing faulty shared memory," *(in preparation)*, 1993.
8. R. P. Kurshan, "Analysis of discrete event coordination," *LNCS*, vol. 430, pp. 414–453, 1990.
9. D. Dill, *Trace Theory for Automatic Hierarchical Verification.* MIT Press, 1989.
10. M. Hennessy, *Algebraic Tehory of Processes.* MIT Press, 1988.
11. R. Milner, *A Calculus for Communicating Systems (volume 92 of LNCS).* Springer-Verlag, 1980.
12. N. Lynch and M. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proceedings of 6th ACM Symp. on Principles of Distributed Computing*, pp. 137–151, 1987.
13. C. A. R. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.
14. R. P. Kurshan, *Automata-Theoretic Verification of Coordinating Processes.* UC Berkeley Lecture Notes, 1992.
15. R. P. Kurshan and L. Lamport, "Verification of a multiplier: 64 bits and beyond," *preprint*, 1993.