National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents

NL 339 (r 88/04) c

Canada

An Efficient Algorithm and Software Package for Verifying Delay-Insensitive Systems Using a Partial-Order Model.

Lin Jensen

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

November 1991

(c) Lin Jensen, 1991

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-73700-X

Canada

ABSTRACT

An Efficient Algorithm and Software Package for Verifying Delay-Insensitive Systems Using a Partial-Order Model.

Lin Jensen
Concordia University, 1991

Networks of delay-insensitive processes implementing a requirements specification are verified using a representation based on partial-order semantics. Processes are represented by behavior automata, with branching and recurrence structure represented explicitly.

The verification algorithm is described. It examines all the distinct actions in the system produced by coupling the specification mirror to the implementation; it effectively constructs a behavior automaton for this system. The algorithm avoids state explosion by performing causality checking (arrow checking) for safety and liveness.

A computer program implementing this algorithm is presented. It has been tested with two traditional asynchronous-circuit benchmarks, namely chains of buffer elements -- which are determinate systems, and rings of DME elements -- which are non-determinate systems.

The complexity of the algorithm depends on the structure of the system being verified (more precisely, on the size of the constructed system behavior automaton). Good clear structure in process specifications is crucial to obtaining good performance.

Empirical results are presented for the benchmarks. Verification cost for the buffer series is $O(n)$ in time and space. The DME series verification cost is $O(n^2)$ in space, and also $O(n^2)$ in time with an ideal implementation. In both cases, n is the number of components in the implementation. This good performance makes essential use of the pre-identified branching and recurrence structure of processes.

Future extensions of the algorithm to quasi-delay-insensitive and delay-constrained systems are sketched.

Key words: Delay-insensitive system, state explosion, partial-order representation, state encoding, verification, causality checking, POM system.

ACKNOWLEDGEMENTS

I would like to thank my family for great patience and understanding during this research and during the writing of the thesis; especially the many hours that I was at home but not available.

Thanks to Dr. Peter Grogono for introducing me to the Trilogy language as part of a course in "Recent Paradigms in Computer Languages." He provided both general and specific encouragement.

Primarily I want to acknowledge the tremendous contribution of Dr. David Probst to this project. I met him at the very beginning of my graduate studies at Concordia, at which time he got me started on the right foot. He was instrumental in choosing this project at the intersection of our interests, and guiding it thro gh its many stages. We had countless meetings and discussions that shaped and then polished the work, right down to a careful reading for run-on sentences. In particular he was very gracious on the question of using execution states instead of behavior states in the final program.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

- - ->	noncausal successor arrow
———>	causal successor arrow
═══>	chain of (one or more) causal arrows
o	socket in command of behavior automaton
A ⟦	dfsm label, start of command
dfsm	deterministic finite-state machine
Γ	a partial order
Ω	successor relation (transitive reduction of partial order Γ)
μ	arrow-labeling mapping in a labeled poset, also label of the 'message' arrow in buffer
σ	socket 'plug in' map for behavior automata, also label of 'space' arrow in buffer

CHAPTER 1. INTRODUCTION

1.1 Motivation

VLSI technology has reached the point where it is no longer possible to ignore the delays introduced by wire connections. In particular, if circuits are synchronized by a global clock, variable delay in propagating the clock signal is called clock skew. The clock must be timed so that the slowest component on the chip has produced its output before the next clock tick. The full speed of faster components cannot be utilized. It is difficult to make improvements to a design because the whole package must be retimed, and changes in placement of components can affect the timing.

Asynchronous circuits are designed to operate without a clock. They have the potential for faster operation, as once an output is produced, it is available to the next component, without having to wait for the clock to tick. Delay insensitivity is a protocol designed to allow asynchronous circuits to operate correctly regardless of speed or delay. This makes correct operation independent of timing considerations, and allows improvements to be made to parts of a chip without affecting correctness, while increasing efficiency.

Existing methods, essentially based on interleaving semantics, for verifying concurrent systems suffer from a 'state explosion problem', which arises from the fact that

they consider all interleavings of the concurrent events. We have been guided by the belief that concurrent systems are less constrained and therefore inherently simpler than sequential ones.

The correctness problem for a circuit, relative to its specification, encompasses two classes of properties, safety and liveness. Safety properties assert that nothing bad ever happens. Examples of safety properties are: a process receives input before it is ready to accept it. This might be when the process has not yet finished processing the last input. If the process is a wire that has not finished sending its last input, this is called transmission interference.

Liveness properties state that something good eventually happens. They can be further divided into progress and fairness. A progress property asserts that an enabled output will eventually occur. A fairness property asserts that each outcome of an enabled conflict, if this conflict is repeatedly enabled, will eventually occur.

1.2 Scope of the thesis

The present work consists of modeling concurrent processes using partial orders. The focus is on hardware processes, such as used in VLSI, at just above the gate level of abstraction. In particular we focus on asynchro-

nous, delay-insensitive processes, as they exhibit a regular pattern of handshaking that simplifies verification. Few delay-insensitive systems are made directly from gates, in particular AND and OR gates are not suitable as delay-insensitive components. Some simple delay-insensitive components are XOR gates, toggles, and C-elements.

An implementation is correct if it meets the requirements specification. When there is asymmetry of control, the requirements specification can be divided into an intra-process part, for actions under the control of the process, and an interprocess part, for actions under the control of the environment. In addition, (the components of) the implementation have interprocess safety requirements that must be observed. (A component cannot make liveness demands on its environment.) Thus correctness means that all the safety and liveness requirements of the specification are met, and also all the safety requirements of processes in the implementation are respected.

A verification algorithm and a computer program implementing it are presented. It checks a network of delay-insensitive processes against a requirements specification, verifying all safety requirements, and the liveness requirements of the specification.

This algorithm can be extended to any systems in which safety and liveness can be specified in terms of causality.

These include quasi-delay-insensitive [4] and delay-constrained [16] systems.

1.3 Related work

Udding [9] models concurrent systems with trace theory. He gives rules for delay insensitivity, and puts forward the 'foam-rubber wrapper' idea that a delay-insensitive process should be indistinguishable from the process with the addition of wires to its ports. Trace theory uses interleaving semantics, which has the drawback of introducing many different traces which do not differ in their partial order. It models concurrency as non-determinism. This leads to a very interesting proof that there does not exist a delay-insensitive fair arbiter. Fortunately, this proof does not carry over into partial order semantics.

Dill [8] also uses trace theory. He develops and implements a theory of automatic verification for speed-independent circuits. This is a wider class of circuits, allowing silent transitions, such as exhibited by AND and OR gates, by making the assumption that wires are instantaneous. This of course requires that components in a speed-independent grouping be placed in close proximity.

Dill makes the important points that the description of the behavior and requirements of a circuit, and of a specification, should use the same formalism, and that the

mirror of a specification can be composed with a (proposed) implementation, presenting the worst possible environment in which the circuit should be expected to operate.

Systems may have many states, particularly when there is concurrency and nondeterminism. If verification is state based, in the sense that you try to look at all the states, and keep a table of the states you have found, you may need lots of memory. For example, whenever there are n concurrent events, there are exponentially many states (2^n) since the events may occur in any order. The key to effective verification using partial orders is to not visit every state.

Partial-order approaches to concurrency include Petri nets, event structures, pomsets, and domains (lattices). Nielsen, Plotkin and Winskel [1] demonstrated an equivalence between occurrence Petri nets, event structures, and domains. Pratt [2] proposed partially-ordered multisets (pomsets) as models for concurrency. A non-determinate process may be modeled as a set of pomsets, which are the possible computations. A slightly different model is an event structure, or pomtree, where the choice structure is explicit. A determinate process reduces to a single pomset either way. See [2] for the definition of pomset.

Probst and Li [4] apply pomsets to the specific problem of specification and proof of correctness of delay-insensitive systems, upon which the present work is directly based.

The most important ideas in their paper are: explicit representation of choice, and finite pomset grammars to capture the recurrence structure of a process. They also extend this to speed-independent systems, a class of quasi-delay-insensitive systems in which isochronic forks allow one process to handshake on behalf of another.

1.4 Overview of the thesis

Chapter 2 presents process modeling using partial orders.

Delay-insensitive asynchronous processes, whose correct operation is independent of timing considerations, obey well-behavedness rules. One necessary condition for delay insensitivity is a regular pattern of handshaking. Processes produce output that gives evidence of proper assimilation of inputs, after which the inputs are enabled again. Handshaking guarantees that safety can be expressed by causality, and causality is crucial to our method.

Causality checking can also be used with other classes of systems, such as quasi-delay-insensitive [4] (speed-independent) and delay-constrained [16] systems.

Our approach to process modeling is to construct behavior automata. We start by identifying the conflict resolution and recurrence structure. This can be drawn first as a 'stick figure' (dfsm). Second, each stick is expanded

to a command, which is a poset of leading sockets and actions. In each poset we only draw the successor arrows, that is the elements of the transitively reduced partial order, and label them. Choices are always visible at the beginning of commands as disjoint sets of leading actions. At the nodes where the sticks join we have non-sequential concatenation rules, defining which actions can "plug in" to sockets. Finally, we reduce the number of arrow labels by discovering arrows between commands (from sockets) that can be equivalenced while preserving the semantics of the process. These are generally arrows which are involved in forwards or backwards conflict. All actions give arrows that lead out from them, including actions at the trailing edge of commands (the arrows come from sockets).

Any traversal of a behavior automaton gives a possible computation, or behavior, of the process. In a behavior, actions are repeated. Each repeated action is a distinct event. I drink many cups of tea over time, each is a distinct and unique instance of the action 'drink a cup of tea.' The events, labeled by action and with their partial order, are thus a partially ordered multiset, or pomset.

We will need a dual focus, on actions to carry out finite verification, and on events to argue about comprehensive coverage and resolve questions about to which event an action refers. (See the buffer examples.)

A consistent cut across a pomset (or branch of a pomtree) represents a present, dividing past from future. The past is the history [3] of the cut. Histories can be equivalenced. Recording the complete history means no equivalence classes at all. The coarsest equivalence relation that preserves semantics is machine state. Different histories are equivalent machine states if they have the same possible futures. In such a case it doesn't matter to the operation of a machine how the state was reached.

The goal of any state encoding is to specify what events are currently enabled, what the conflicts are, and how this state could have arisen.

The state corresponding to any cut can be encoded by the set of arrow labels in the cut. At the same time, this encoding of state does extra work for us, because the arrows specify the non-sequential relationship between past and future events. They are the proper glue for rejoining pomsets that have been cut apart with scissors. This is the novelty of our state encoding.

The coarsest possible equivalence relation on arrows (that does not alter the process) gives history-free execution states, equivalent to a reduced fsm. We have explored a finer relation, called behavior states, with some recent history included. This has advantages and disad-

vantages. In the end, the behavior-state program was simpler and the argument for comprehensive coverage was direct, but it took more time and used more memory space.

Finally, chapter 2 concludes with a streamlined encoding useful to the automatic verifier. With individual actions there are arrows in, arrows out, and 'marks' on the (co)initial actions of commands. These marks record recurrence, branching, or both. They encode the structure of the stick figure.

Behavior automata have more structure than Petri nets, because conflict and recurrence are explicit. They define an intermediate position between free-choice Petri nets and general Petri nets.

Chapter 3 contains some examples of process specification. In particular, buffers, arbiters, and DME elements are modeled.

Chapter 4 introduces networks of processes. In open networks, some output ports are connected to input ports, while some remain external, to be connected to the environment. For verifying, a closed system of processes is formed by connecting a network to the 'mirror' of the specification, which acts as a conceptual implementation tester, driving the network of implementation processes. With such a closed system, there are two partial orders, one given by the causal arrows of all the processes in the system, the

other by all the noncausal requirements arrows. Checking whether the implementation meets the requirements specification is now turned into checking simple relations in the closed system.

In chapter 5, correctness of such a system is defined in terms of these successor arrows. For safety the correctness condition is that all the noncausal arrows are supported by chains of causal arrows, or that the noncausal order is contained in the causal order. This can also be expressed as a form of language containment: The language of the causal order is contained in the language of the noncausal order, not to be confused with the more usual form in which the language of the implementation is contained in the language of the specification.

Wires are processes which may be included in delay-insensitive systems. In this chapter we show that wires may be added to such a system without changing the system correctness.

Chapter 6 gives two main examples of system verification which are delay insensitive. These cover producer-consumer and mutual exclusion problems, since these are arguably the main synchronization problems in concurrent systems.

The first example is a class of deterministic buffers, followed by input non-determinate buffers, with data in FIFO

order. In these examples we encounter instances of more than one arrow of the same name crossing a state cut.

The second example is a non-determinate network consisting of a ring of DME elements implementing an arbiter. In tracing a verification algorithm for this network, I noticed a problem of livelock, namely, the possibility that the ring would continue to pass the token around without granting user requests.

Chapter 7 presents the verification algorithm, and demonstrates both that it does terminate and that complete causality coverage of the system under verification is obtained.

The verification algorithm creates a closed system of processes by taking the mirror of the specification, which then drives the closed system. The mirror is obtained by interchanging the role of input and output ports. The system is then explored. It has two orders, causal and noncausal. All distinct system actions are visited. Checkpoints are taken at a small number of system states corresponding to system causal choice, or the start of recurring commands of the specification, or both. To keep the number of checkpointed states small, we defer visiting 'marked' events until all unmarked (plain) events have been visited. Verification continues until all choices have been explored and no new checkpoints are produced.

Safety and liveness properties at states not actually reached during execution of the algorithm are verified by arrow checking. For every noncausal arrow (safety requirement) in the system pomtree, we must find a chain of causal arrows that supports it. This guarantees that there is no reachable state at which an event has occurred without its required preconditions having occurred. For liveness (of the specification), we also require that the causal preset of an event exactly matches its noncausal preset. In other words, the specified (output) event must occur following just its enabling inputs, without needing additional inputs.

The verification is not a simulation. We move through all the system actions in order to verify requirements at each action. There is no chance involved.

Chapter 8 describes the program in Trilogy that implements the algorithm. It either reports an error detected, or terminates successfully after completely verifying the system.

Chapter 9 presents results of running the program with the examples from chapter 6 used as test data. It also contains a discussion of the tradeoffs between using behavior states, containing some recent history, and history-less execution states.

Chapter 10 gives some conclusions and suggested directions for further work.

Finally, three appendices contain a user's guide to the programs, sample program output both for successful verifications and runs where bugs were found, and the program listings.

CHAPTER 2. PROCESS SEMANTICS

2.1 Introduction

In this chapter we motivate and present our representation of processes, and their semantics. A reasonable set of well-behavedness rules for hardware processes is given, including rules for delay-insensitive processes. A reasonably broad class of processes can be specified with an explicit choice and recurrence structure, using a finite set of actions.

A behavior is a run of a process. It contains events, some of which are concurrent, resulting in a partial order on the events. In contrast to fully-ordered traces, we are not concerned with the accidental order of concurrent events.

The transitive reduction of a partial order gives its successor relation. The elements of this relation are successor arrows. A consistent cut through a partial order contains some successor arrows. It is reasonable to label the arrows, as a way to encode state, and to specify the non-sequential relation between past and future events.

We formalize process representations as behavior automata. Arrows between actions are labeled. Behavior automata have boundaries at loop and choice cutpoints which explicitly encode the recurrence and choice structure. Their commands can be effectively hooked together by labeled arrows. Then a behavior automaton can be represented by a

table of atoms of the form: arrows in, action, arrows out, 'mark' if on cutpoint. This forms the basis of our data structure for processes.

2.2 Partial orders, successor arrows

This section motivates the partial-order description of processes. We consider events forming a pomset or pomtree.

A process consists of a collection of ports, at which events can occur. The history of a process consists of some events at ports that are initially enabled. When events occur, they enable other events, and so on. This gives a partial order of the events. If the events are labeled by a finite alphabet, for example, by their port names (1), this becomes a partially ordered multiset, or pomset. [2] If there is non-determinism the pomset branches into a pomtree. When choices are made, the future behavior of the process lies entirely on one sub-pomtree. Equivalently, a non-determinate process gives rise to a set of pomsets, each of which is a behavior (possible computation).

We draw pomsets and pomtrees by labeling the events, and connecting them with successor arrows. The successor arrows represent enabling. An event e (that has yet to occur) is enabled precisely when all the events with arrows leading to it have occurred.

---

(1). Later we will see reasons for refining this labeling.

2.3 Consistent cuts

Consider a pomset representing some computation, in which we have drawn the transitively reduced partial order as arrows. There are many possible divisions of a set of events into the past (those that have occurred) and the future. A cut drawn between the past and future events will cut across some arrows. It is consistent if all the arrows cross in the same direction. Otherwise it cannot represent any present, as there would be future events preceding past ones. Figure 1 shows some consistent cuts across a pomset.

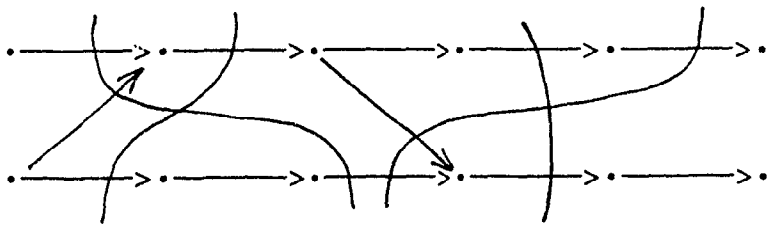


Figure 1 Consistent cuts

Given any consistent cut, we have two pomsets, past and future. The "bridge arrows" that cross the present are part of neither. We do not throw them away, since they convey important information. In particular, they specify the exact temporal relationship between past and future events, also they encode precisely which events are currently enabled. An event is enabled at present iff all the arrows leading to it are in the cut.

2.4 Asymmetry of control

There are two kinds of arrow, those that make up the inter-process protocol and those that make up the intra-process protocol.

At the hardware level, ports are either input or output ports. A process has control over its output, but the environment has control over input. Port events consist of transitions. Two transitions may not occur concurrently at the same port. Events at a port therefore consist of a sequence of electrical transitions. Alternate transitions are of necessity electrically opposite, but no meaning need be placed upon the sign of the transition [5].

Given this asymmetry of control, we speak of safety at input ports. If a transition arrives at a port when the process is not ready to accept it, the process' behavior becomes undefined.

Given asymmetry of control, we underline output actions, and draw the arrows using two distinct styles.

We draw dashed arrows leading to input events. We call them noncausal arrows, forming the inter-process protocol, since input events are under the control of the environment. Thus dashed arrows are safety requirements. Each requires the environment to wait until the preceding event has occurred before providing the input.

We draw solid arrows leading to output events. We call

them causal arrows, forming the intra-process protocol, since output is under the control of the process. They specify when the process can produce output. If one specifies that an output is required (see rule 5), they become liveness (progress) obligations: the process is required to produce this output once all its preconditions, as indicated by the solid arrows, have occurred.

2.5 Delay insensitivity

Asynchronous, delay-insensitive circuits are designed to operate correctly no matter what the delays are in producing outputs or transmitting them over wires. Wire delays are taken seriously. Thus the introduction of a wire into a circuit is modeled as a process in its own right. Oddly enough, this does not introduce additional complexity, because the rules for delay insensitivity ensure that whenever a system is safe, the safety of the wires is ensured, so the wires do not need to be explicitly taken into account after all. A proof of this is given in section 5.5.

The well-behavedness rules for delay insensitivity require handshaking between processes. The simplest delay-insensitive building blocks often have speed-independent implementations at the gate-level. A speed-independent process is a form of quasi-delay-insensitive process in which the assumption of isochronic forks allows one gate to

handshake on behalf of another that remains silent. By specifying these 'aliases' one can use the verification methods developed in this thesis also at the gate level.

2.6 Well-behavedness rules

A basic well-behavedness rule forbids auto-concurrency:

Rule 1: Two events at the same port may not be concurrent. They must be separated by at least one event at some other port.

In general, auto-concurrency is forbidden. For delay-insensitive processes, this is enforced by handshaking:

Rule 2: A process cannot require a direct ordering of its input events. Two input events which are not concurrent must be separated by at least one output event.

This rule applies both to events at the same port and different ports. With regard to a single port, we must ensure that the first transition has been properly received and assimilated before another can occur. With respect to different ports, we do not want to assume time constraints on either transmission media or receiver logic [14].

Receivers may disagree about what precise point in a monotonically rising (or falling) voltage transition will be recognized. This means that even if we know the order in which signals are produced by the environment, this order cannot be guaranteed to be recognized by the receiving process.

So in delay-insensitive processes, noncausal (dashed) arrows always lead from output events to input events. They represent safety requirements demanded by the process of its environment. The preceding output events are permissions that the environment must respect.

From this it follows that there is not much point in specifying any direct ordering between output events (at different ports). The only time this could be convenient would be following a non-determinate choice. Even in this case, other processes may not make use of such ordering, and such a specification cannot be mirrored to produce a driver for testing its implementation.

Delay-insensitive closure removes the output ordering, since it would not be preserved over wire connections, and is therefore unobservable by the environment. The ordering between output events is simply removed from the partial order; this leaves the ordering with preceding input events, which now appears as arrows in the transitive reduction.

Rule 3: Two output events at the same port must be separated by at least one input event.

This input event allows the environment to signal that the first output event was properly received. Rule 3 is slightly stronger than rule 1.

Rule 4: No cross-disabling. No event can disable a previously enabled event of the opposite type.

For example, once a process has an output enabled, permission for that output cannot be withdrawn by the arrival of an input signal, such as an inhibit.

In restricted process theory, output liveness is assumed:

Rule 5: When an output event is enabled, there is no choice between producing that event and not producing it. There would only be a genuine non-determinate choice between producing alternate outputs.

Causal (solid) arrows now represent progress requirements. They usually lead from input to output events.

2.7 Finite-state processes

Since we are applying these methods to circuits, we consider only finite-state processes. If we think of state as dividing events that have occurred from those that have not, a state is associated with a consistent cut.

Rule 6: Processes are finite state. In particular, there are a finite number of ports, a finite number of arrows leading from and leading to any event, and an upper bound to the number of arrows across any consistent cut.

The last part follows from the finite state requirement. It rules out, for example, infinite buffers.

2.8 Reasonable, visible choice

We rule out unreasonable coupling of concurrent choices:

Rule 7: Concurrent choices with concurrent, disjoint preconditions must be independent.

Otherwise it is possible to specify coupled choices that must be remembered without bound, much like an infinite buffer. An example of a violation of rule 7 is given in

section 3.1.

It is important for choice to be visible. If a process makes a hidden choice, this can dramatically increase the complexity of verification. We require that any choice is visible at the ports of a process; what you see is what you get.

Rule 8: When a process has a choice, it is only between disjoint sets of enabled actions.

2.9 Finite representation and recurrence

We restrict ourselves to processes with the recurrence and branching structure given explicitly. We describe them with behavior automata. Here the development of this idea is sketched, leading to a formal definition given later. A behavior automaton is a collection of finite poset commands, and rules for concatenating them. Each command is a partially ordered set of actions. The infinite pomtree of a process can be constructed from its automaton by starting at the initial state and growing it by adding on commands wherever allowed.

We want behavior automata to be as compact as possible, and in a preferred form, in which choice cutpoints are at the beginning of commands. These may also be loop cutpoints, whenever commands end in a recurrent choice. There may also

be determinate loop cutpoints, from which a single command starts. The choice of loop cutpoints is somewhat arbitrary.

We can draw a stick figure in which the arcs represent the commands, and the nodes are choice cutpoints, loop cutpoints, or both. Formally this is a deterministic finite state machine (dfsm), since when two arcs leave a node, they are labeled by distinct commands. In fact, the purpose of the stick figure is to make explicit the non-determinism. Choice is visible in the stick figure as more than one command leaving a node. This is forwards conflict, real non-determinism. Backwards conflict is also visible as more than one command arriving at a node. This is simply the condition where a state can be arrived at in different ways; different pasts may produce the same state. The chosen cutpoints have in fact been singled out because of the forward or backward branching. In the most reduced form of a behavior automaton, each node represents a different execution state.

The elements of commands are not events, but actions that repeat to produce many events. We can give each action a distinct name, this results in a finite alphabet of actions. We will use the port name and primes to label actions. This is more general than just using the sign of the electrical transition. For example, there could be a 6-cycle protocol in which each three successive actions at a port were semantically different.

2.10 Labeling successor arrows

The function of a state encoding is to specify which actions are enabled, which are concurrent, what conflict exists, and what past actions could have enabled these. This section presents our novel method of labeling arrows, that leads to a state encoding that does all this and also specifies the non-sequential relationship between past and future events.

We label all the successor arrows in a behavior automaton. The arrows that join commands require particular attention whenever there is backwards or forwards conflict. Informally, give the same label to arrows with the same semantics. Formally, do this providing the resulting pomtree is unchanged.

State is encoded as the set of arrow labels in the current cut. (2) It is possible to name all states in this same way, not only the nodes of the behavior automaton, but also any state we might encounter. From this state encoding we can determine the currently enabled actions, since an action is enabled precisely when all the arrows leading to it are in the state.

---

(2). Further equivalence would be possible. Conceivably a pair of arrows, (say 7 & 8) could be equivalent to a single arrow (say 5).

2.11 Behavior automata

Here is the formal definition of behavior automata. The conflict resolution and recurrence structure is given by a 'stick figure' (dfsm). There is an alphabet of actions, Act, and one of sockets, Soc. Act $\cap$ Soc = ϕ. Each stick is expanded to a command, which is a labeled poset of leading sockets and actions. We draw as successor arrows the transitively reduced partial order, and label them.

Formally, a labeled poset is a triple (B, Γ, μ), where (i) B is a subset of Soc U Act, (ii) Γ is a partial order over B, and (iii) μ: Ω --> Arr. Ω is the transitive reduction of Γ. Arr is the alphabet of arrow labels. Nothing may precede a socket in a command. An action is initial in a command iff no other action precedes it.

There is a mapping σ: Soc --> P(Act) between sockets and the power set of actions that may plug into them. In the case of backward conflict, the image set is not a singleton.

Choices must be visible at the beginning of commands (see well-behavedness rule 8). Specifically, if more than one command leaves a node, the sets of initial actions should be disjoint. At the nodes where the sticks join we have non-sequential concatenation rules, defining which actions can "plug in" to sockets. To minimize the automaton, reduce the number of arrow labels by discovering arrows between commands (formally, arrows from sockets) that can be

equivalenced while preserving the semantics of the process. These are generally arrows that are involved in conflict. In particular, since the states at which commands join are specified by arrow labels, whenever sockets in two commands plug in to the same action, arrows leaving them should have the same label.

It is quite possible for arrows to pass by (punch through) commands. As a pomtree is grown by adding commands, there will always be some "latest" events, with nothing following. When adding a new command, there is no requirement that all the latest events plug into sockets of the command, or that sockets fit events in the most recent command. Each socket fits the most recent event allowed. A socket may not fit an event that has already been plugged into another socket with the same arrows. (3)

A behavior automaton has an initial action, INIT (or reset). It is the start node of the dfsm. All the sockets of any command leaving this node plug into INIT; sockets of later commands that may be filled by some set containing INIT take the most recent element in the set. INIT is the support for well-foundedness. The state immediately following the reset action INIT -- more generally, an arbitrary partial execution -- is a function of the state encoding that has been constructed.

---

(3). Later we deviate from this rule when introducing "tickle" arrows.

A behavior automaton is a tuple (D, C, Φ, σ), where D is a dfsm, C is the set of commands, Φ is a one-to-one mapping from commands to sticks of D, and σ is the mapping from sockets to sets of actions.

The infinite pomtree of the process is generated by starting at INIT and traversing the behavior automaton (infinitely), branching whenever more than one command leaves a node.

Figure 2 compares standard FSM and behavior automaton pictures of a wire. Note that the nodes and arrows are reversed. In the pomsets, states occur between events, and are thus related to the arrows. Each arrow gives partial state information. See p. 17 for graphic conventions.

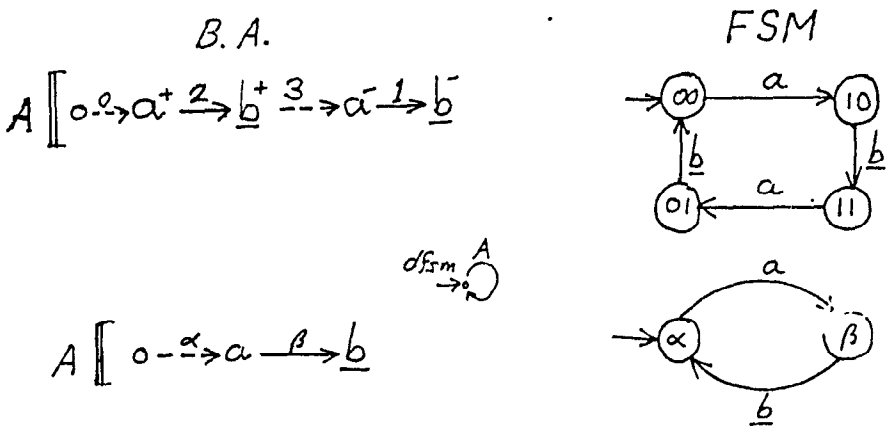


Figure 2 Wire processes - 4-cycle & 2-cycle

Note that state is relative to the alphabet of the process. A (digital) wire can be considered as having four states or two states (sending and quiescent) depending on whether the voltage level is important. The first interpre-

tation corresponds to a four-cycle protocol, and the second to a two-cycle protocol, as used in transition signalling [5], in which rising and falling transitions are considered equivalent. However, for composition of processes, it is generally important to specify the electrical nature of initial states.

Figure 3 shows the behavior automaton for a C-element, with rising and falling transitions equivalenced. There are 4 different arrow labels, concatenation of ıe command is by joining up the partial arrows at the end of one to the beginning of the next. There are 4 different cuts through the whole infinite behavior of the C-elc nent. They can be encoded as sets oı the arrow names. These are {1,2}, {1,4}, {2,3} and {3,4}. The initial state is enccded the same way; it is {1,2}.

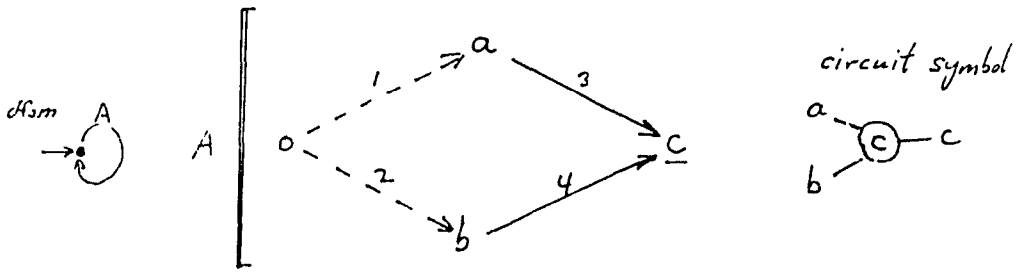


Figure 3 C-element

2.12 Equivalence of behavior automata

Behavior automata give a compact representation for a

class of processes we are interested in. They have additional labeling that is not needed in a full pomtree. Let's be clear that the pomtree is the process, and only the labeling of ports has physical meaning. Thus any transformation of process encoding is correct if it preserves the shape of the pomtree.

Definition: Two behavior automata are equivalent iff they yield the same pomtree, where arrows are not labeled and events are labeled by port names only.

As a simple example, a wire, whether encoded with a two-cycle protocol or with a four-cycle protocol, is still the same process.

This concept has two consequences. It allows us to reduce a specification to minimal sets of actions and arrow labels. On the other hand, it justifies 'fleshing out' a process to match its use in a network, for example, if a wire used in a network where a four cycle protocol is used, one can choose its four-cycle representation.

2.13 Streamlined encoding of behavior automata

Next we eliminate sockets in order to streamline the eventual concrete data structure. Conceptually, a behavior automaton is the dfsm with the sticks expanded into the commands. Actions that plug into sockets in effect have arrows leaving them, according to the mapping σ for action:

socket matching. As a result, all actions now have arrows leaving them, so the state after any partial execution has a state encoding using arrow labels.

Choice and recurrence can be encoded as follows: Each initial action of a command is 'marked' as a loop or choice action. An action a is initial if there is no action x in the poset such that $x < a$ according to the partial order Γ. It is possible to have co-initial actions in a command, this fact must be additionally recorded. An initial action is a choice iff more than one command leaves the node. By rule 8, the choice is visible, meaning that the sets of initial actions in conflicting commands are disjoint.

Choice means conflicting enabled actions, in other words, forwards conflict. Forwards conflict is an arrow with multiple destinations. Backwards conflict is an arrow with multiple sources. Note the symmetry between forwards and backwards conflict.

The initial state, which immediately follows INIT, can be specified by arrow labels, just like all other states are. It contains all the arrow labels that are allowed to leave from INIT according to the socket mapping σ. It may contain multiple occurrences of some of these arrows, in order to partially enable several occurrences of some action(s). For example, a buffer could accept several successive inputs, until it was full. Of course some other

arrow would have to prevent simultaneous inputs.

Since processes interface with other processes at their ports, it is also necessary to specify the initial logic levels of the ports. Thereafter, transitions at each port are strictly alternating.

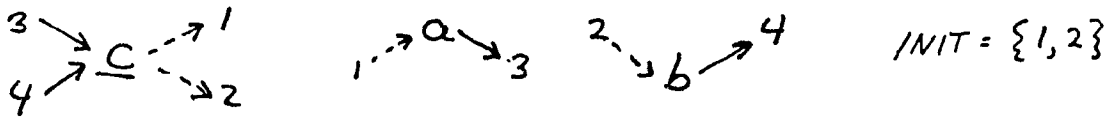
An expanded stick figure is similar to a Petri net, but we have marked the choice actions that follow choice cutpoints (even though they could be deduced from the arrow conflicts), and the actions following loop cutpoints (that cannot). In this way we preserve the structure of the stick figure. We make use of these marks to control the verification algorithm and establish termination and comprehensive coverage. Because of the structure, compact termination tables can be constructed.

2.14 Atomic action rules

To obtain a data structure suitable for representing a process in a computer implementation, construct a table of rules, in which for each action are associated arrows needed and arrows given, and a 'mark' for actions that start a command, and may also be choices. In addition a list of co-initial actions (if any) is needed, and the initial state. This table does not need to be divided into commands, since it can be used to reconstruct them. This encoding must preserve the semantics of the process. In particular the

enabling relationships must be preserved.

Figure 4 shows the atoms of the behavior automaton for the C-element, and the corresponding table of rules. The meaning of (1)**a**(3) is that action **a** needs arrow 1 and gives arrow 3. In addition, a and b are marked as co-initial loop actions.



Rules: (3,4)**c**(1,2), (1)**a**(3), (2)**b**(4)

Figure 4 Atoms of C-element

The semantics of needs and gives are as follows:

Needs: An event is enabled at a state iff all needed arrows are present, and when the event occurs, the arrows are used up, i.e. not present in the new state.

Gives: When an event occurs, these arrows are added to the new state.

Arrow naming must preserve the enabling relationship. In terms of the table of rules, conditions for the proper naming of arrows are: No arrow should appear in the needs list of more than one rule, unless it represents a non-

determinate choice between those actions. No arrow should appear in the gives list of more than one rule, unless it represents a backwards conflict. (Note that the same arrow can do both -- cf. the arbiter.) There is some flexibility here. If we wish our states to be 'history-less' execution states, we make sure that when the execution state is the same (i.e. same possible futures), the arrow labels are the same. If we wish to use the notion of behavior state [15], we can give all the 'backwards conflict' arrows different labels corresponding to their different sources, so that none of them appears in the gives list of more than one rule.

Choice on this level is now seen as the same arrow label needed by more than one rule. Likewise, backwards conflict is seen as the same arrow label given by more than one rule. We can refine our notion of (repeated) action to be: same port, same rule.

In the examples we have considered, equivalenced arrows appear to be essentially the same as backward and forward shared places in Petri nets, which is not surprising in view of the correspondence between Petri nets and event structures shown in [1]. Non-equivalenced arrows can be modeled with colored tokens.

CHAPTER 3. PROCESS EXAMPLES

This chapter contains examples of our process encoding, starting with simple examples, and including buffers, arbiters, and DME elements (which will be used to implement arbiters). The last two are (output) non-determinate. Buffers have the interesting property of allowing several arrows of the same name in a state, making the state encoding a multiset.

3.1 Simple process examples

Figures 5-7 give 3 examples of pomsets of determinate processes consisting of 2 lines (chains of solid and dashed arrows) and varying degrees of synchronization between them. The commands are boxed. They have been labeled according to the above algorithm. Figure 5 is simply 2 wires. A cut is shown that is equivalent by label to the initial state, and intuitively the same machine state, even though events have not progressed completely through the first command. This process as a machine is the cartesian product of the two component wires, with 4 states. It is not strongly synchronized, since progress on either line is independent of the other.

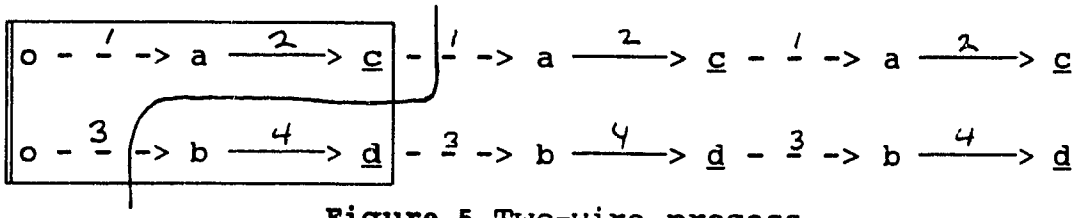


Figure 5 Two-wire process

Figure 6 contains additional arrows going only from

the top line to the bottom line. This corresponds to an infinite buffer, since the top line can advance arbitrarily far ahead of the bottom line. Thus it violates well-behavedness rule 6. There can be many μ-arrows in a consistent cut (state). In this case it would seem that using integers to record the number of each type of arrow would adequately encode the state. Just two remarks at this point. The arrows representing the handshaking required for delay insensitivity (seen here in the two lines) cannot be more than one in a state (because they occur in lines). Later we will see an information-containing FIFO buffer in which the order of some of the multiply occurring arrows should be part of the state. With real data values that can be buffered, the order of output matters.

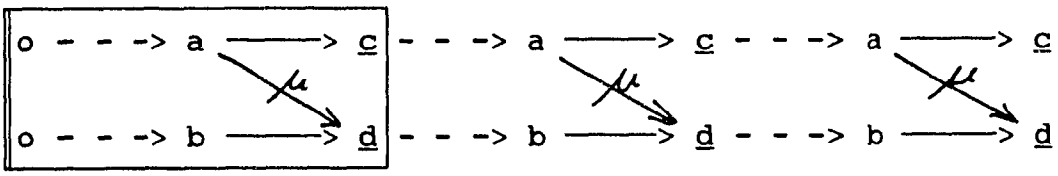


Figure 6 Infinite buffer pomset

Figure 7 shows a third possibility in which there are arrows both up and down, making this process strongly synchronized. Note that the three occurrences of each port action are different (although it is hard to speak of semantics when I don't know what this process is for, rather like an assembly language program with no comments). The arrow labeled β', for example, looks 'locally' the same as β, but

is part of a sequence leading up to the obviously different event d'.

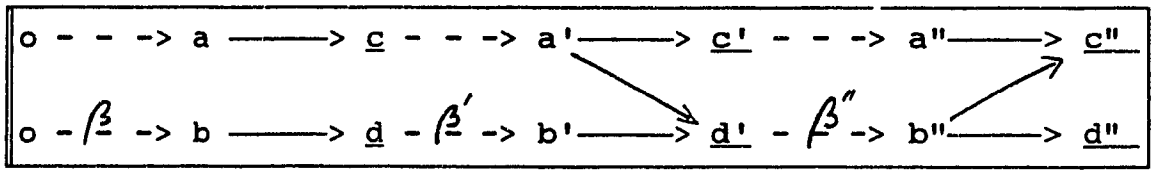


Figure 7 Strongly synchronized 6-cycle process

Also note that since each series (e.g. a, a', a'') repeats, there is no relation to the actual direction of the transitions on ports, which must strictly alternate electrically. On the one hand, we are only interested in the repetition of patterns. On the other hand, when composing processes, it becomes necessary to agree on the electrical nature of the initial state of ports.

Well-behavedness rule 7 forbids coupling of concurrent choices when there is no precondition in common. Figure 8 shows a case where if a rather that c is chosen, then b rather than d is also chosen. This is quite different from the case of two independent choices. Since all the events on the x line are concurrent with the y line, an unbounded number of choices might have to be remembered. Such a process violates rule 7.

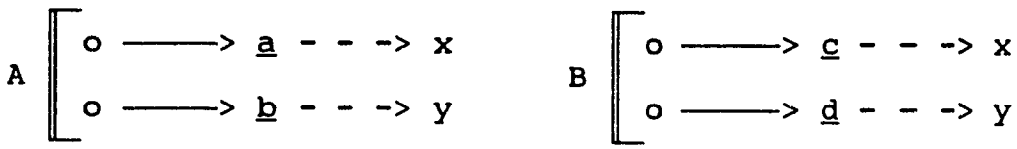


Figure 8 Improper coupling

Figure 9 appears similar, but this time both precon-

ditions must occur before the 'coin is flipped'. We can easily imagine this corresponding to the occurrence of an internal event. This is a well-behaved process. But viewed as a black box, the coupling between the two output events must be explicit in any encoding, to distinguish this from a process where there might be two independent coins.

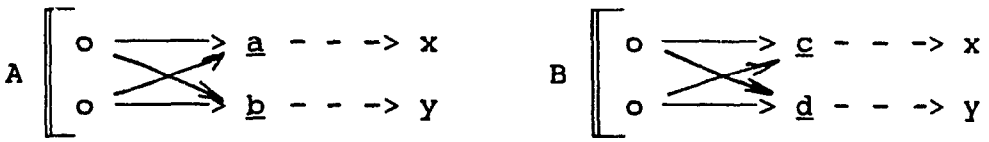


Figure 9 Acceptable coupling

3.2 2-Transition buffer -- 2 versions

This section presents a buffer that requires the environment to guard against over-filling, and then shows how to modify this buffer to provide for internal safety against over-filling. The transformation utilizes delay-insensitive closure to remove some output ordering. An implementation is given in chapter 6.

Figure 10 shows part of the infinite pomset for the 2-element buffer from [4]. In this buffer, the environment is responsible for making sure that the buffer is not over filled. To do this it must monitor both output ports. This is a safety requirement, expressed by the upward slanting dashed arrows. They are all equivalent, and there can be up to 2 of them present in a state.

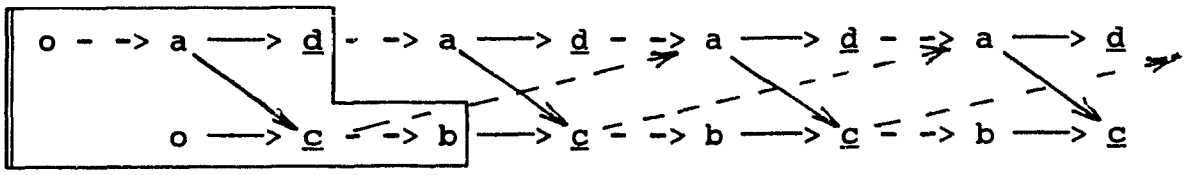


Figure 10 Buffer with external safety

Safety (against over-filling) can be made internal to the process in two steps. First,replace the slanting dashed arrow (there is room in the buffer) with a solid arrow, as shown in figure 11.

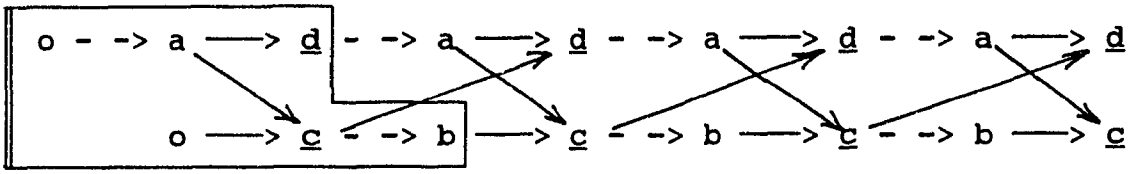


Figure 11 Buffer, safety being made internal

Second, take the delay-insensitive closure from the preceding input events. This gives the external view of the process. The semantics of this new arrow is: "Either there is room in the buffer, or I am in the process of making room, so don't worry." Figure 12 shows the new buffer specification.

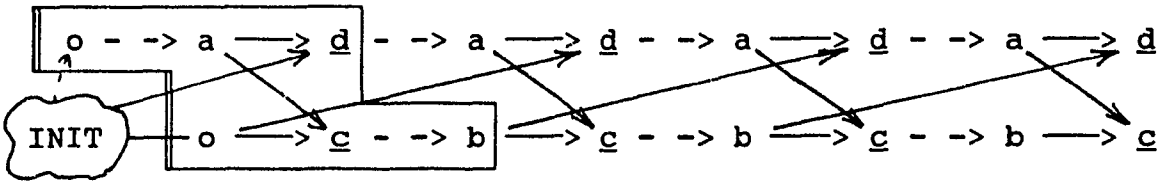
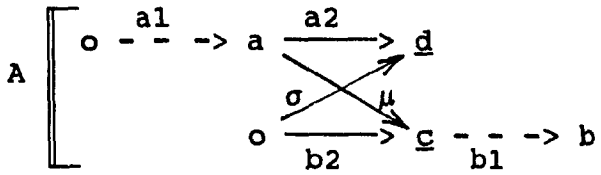


Figure 12 New buffer, after DI closure

Note the initialization, which makes the output interface active. Such buffers may be composed to form larger capacity buffers (no deadlock at the internal interface).

The formal behavior automaton for a 1-transition buffer is given in figure 13.



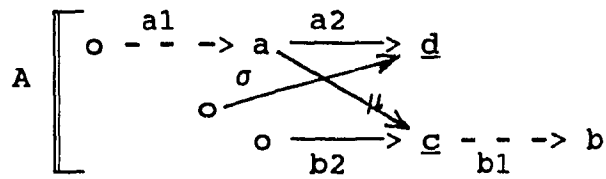
Rules: {a1}**a**{a2,μ}, {b1}**b**{b2,σ}, {b2,μ}**c**{b1}, {a2,σ}**d**{a1}

Figure 13 Behavior automaton for buffer

This specification is generalized to any number of transitions in the next section.

3.3 n-Transition buffer

Figure 14 is a generalized specification for an n-transition buffer. The command is the same, only the initializations are different, in the number of σ arrows. The tabular form also serves for buffers in which both interfaces are passive.



Rules: {a1}**a**{a2,μ}, {b1}**b**{b2,σ}, {b2,μ}**c**{b1}, {a2,σ}**d**{a1}

Figure 14 Behavior automata for general buffers

The socket of the σ arrow attaches to the earliest b in the sequence of b's that does not yet have a σ arrow. If we use as a reference any state in which d is enabled, the σ arrow goes to the i-th b preceding the reference state,

where there are i σ arrows in the state.

3.4 Interfaces

Processes are usually connected together into larger networks. This is accomplished by connecting neighboring processes together through their ports. We can define an interface to be a subset of ports designed to be connected to another process. Interfaces can be used to verify many of the noncausal (dashed) arrows in a network verification. Figure 15 shows the composition of the output interface of one buffer with the input interface of another. All the dashed arrows in each interface are supported by solid arrows in the other.

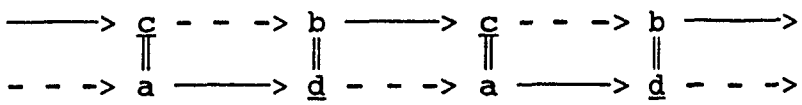


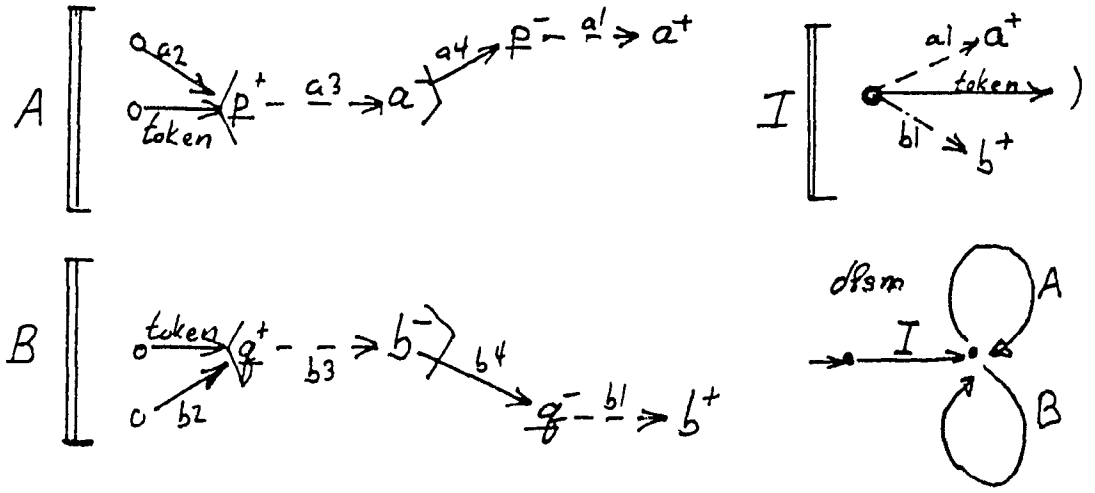
Figure 15 Interface between buffers

An interface projection is a projection of the process' partial order on the events associated with that interface. If in particular the interface projection contains all the noncausal arrows leading to its input events, then safety at the interface can be established by reference to the neighboring process alone. Since typically this is true of many arrows, a program can be optimized by looking first for this simply checked situation.

3.5 Arbiter

An arbiter resolves conflicts over shared resources. Each client of an arbiter follows a 4-cycle protocol: request, <grant, release,> acknowledge. The arbiter ensures that no two clients can be in their critical sections, marked with < >, at the same time.

Figure 16 gives the specification of a 2-way arbiter. There are 3 commands in the behavior automaton: an initial section containing requests leads up to the unique choice state, 2 other commands return to this state after granting one of the requests. The interesting arrow is labeled token. It enforces the restriction that no more than one client can be in its critical section at one time. No possible state can have more than one token arrow, and choice is visible as more than one possible destination for that arrow, namely p^+ or g^+. The source of token is the latest occurrence of INIT, a^-, or b^-, shown graphically by placing these on the middle line.



Rules:
{a1}$\mathbf{a^+}${a2}, {a2,token}$\underline{p}^+${a3}, {a3}$\mathbf{a^-}${a4,token}, {a4}$\underline{p}^-${a1}
{b1}$\mathbf{b^+}${b2}, {b2,token}$\underline{g}^+${b3}, {b3}$\mathbf{b^-}${b4,token}, {b4}$\underline{g}^-${b1}

Figure 16 Behavior automaton for arbiter

There are concatenations in which the token arrow is not transitively reduced, namely when one command follows itself. I choose to keep the token arrow as an account of the process anyway, since it is a semantically important part of the process. It guarantees that critical sections are serialized. Besides, eliminating it in those cases would only add a complication to the verification algorithm.

The partial order is also encoded in a table of atom rules. It is not necessary to separate this table into commands. To recover the individual commands from the table, it is only necessary to record that $\underline{p}^+$ and $\underline{g}^+$ are the initial events in the recurrent commands. The data structures of the verification program record this information, which also

triggers storing the state at the end of a command.

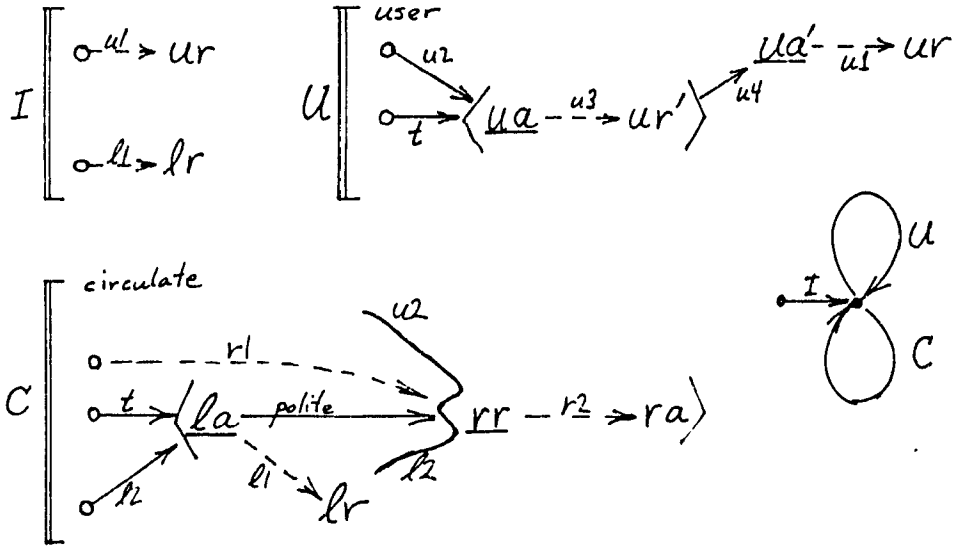
In the list of atom rules it is not necessary to duplicate the actions a and b of the initial command, as they have identical arrows as the same actions in the other commands.

Arbiters with more interfaces are specified similarly. There would be a request from each interface in the initial command, and a new recurrent command for each interface. The program presented later has a procedure to automatically generate a specification for an arbiter of any size.

3.6 DME element

Dill [8] described a ring of Distributed Mutual Exclusion elements which implement an arbiter with as many interfaces as elements in the ring. The DME element itself had been implemented by Martin [6]. It has 3 interfaces, and memory for states of 'having a token' or not. It handles requests for entering a critical section on one interface. The other interfaces are intended for connection with adjoining DME elements. One requests the token; the other gives away the token. When they are connected in a ring, and initialized with exactly one DME element with a token, they implement an arbiter. The token circulates in one direction. Both the arbiter and the DME element are output non-determinate. The ring is described in section 6.5.

Figure 17 is a behavior automaton for a DME element. This follows the description given by Dill [8] except that the neighbor interfaces use a two-cycle protocol in the spirit of Sutherland [5], omitting the second handshake, which Dill says is for the convenience of an implementation.



Rules: {u1}**ur**{u2}, {u2,t}**ua**{u3}, {u3}**ur'**{u4,t}, {u4}**ua'**{u1}
{l1}**lr**{l2}, {l2,t}**la**{u1,polite},
{r1,polite, (u2 or l2)} **rr** {r2}, {r2}**ra**{r1,t}

Figure 17 DME element

The behavior automaton shown is for a DME element initialized with the token. It is necessary to mark the choice actions **ua** and **la**, and specify the initial state as (u1, l1, r1, t). A DME element initialized without a token has initial state (u1,l1,r1,polite) and a different I command, that also includes **rr** and **ra** as in the last half of the C command.

Three interesting points should be made about this

specification. First is the arrow polite (la———>rr), since it goes from an output event to another output event. This is legal, but one cannot expect to observe the ordering from outside. The semantics of the arrow is clear enough: "I have given up the token, so it makes sense to ask for it back. I will be polite and not ask for a second token when I have one." We could take delay-insensitive closure as we did with the buffer, but across a bridge this gets messy. The arrows leading to la are part of execution states that have different possible pasts (correspond to more than one behavior state).

Secondly, the choice is visible up front. The implementation apparently decides which interface to service before it requests the token from the neighbor DME. If the specification includes this hidden state, the cost of verification is greatly increased.

Thirdly, note the following interesting situation. The request for token, rr, should only be made when the DME does not have the token, and there is a request for it either from the left neighbor or from the user. We call these arrows tickle enabling, because the arrows (u2 or l2) are not used up, they are also needed to enable the granting of their respective requests, once the token has been obtained (from the right neighbor).

(One could also change the specification by eliminating

the 'tickle' enabling. But the resulting DME element would always request the token as soon as it had given it up. In such a ring the token would be continually circulating even in the absence of any user requests.)

This is a form of 'confusion' that cannot be cleanly modeled by an event structure. Thus we must extend our model to cover the requirements of this practical example.

The choice semantics for Tickles is a practical one, motivated by the existence of the arrows that are also needed, to enable the granting of requests. Also, there is the problem of scavenging unconsumed arrows, if we produced extra arrows for this purpose.

The semantics of tickles is:

Tickles: Only one of these arrows is needed to enable the event. The arrow is not removed from the state, since it is assumed to be needed for the enabling of another event.

CHAPTER 4. NETWORKS and MIRRORED CLOSED SYSTEMS

4.1 Introduction

Next we discuss the concept of a process or a network of processes implementing a specification. Our strategy is to compose the network with the mirror of the specification to form a closed system. Safety and liveness are then defined as properties of the graph of the "extended" pomtree of this closed system, where "extended" means that there are two partial orders.

4.2 Open network + mirror = closed system

Our representation of a process is a requirements specification. It can also be used to characterize a proposed implementation, or part of it. Processes can be connected together to form a network that is the implementation. A single process implementation is just the simplest case of a network; we develop our algorithm for networks in general. Such an implementation network is an open system. Some network output ports are connected to other input ports, while others remain external, corresponding to ports of the specification.

Our strategy is to form a closed system by taking the mirror [8] of the requirements specification, and composing it with the network. The mirror is the implementation tester. It provides the worst possible environment in which the network can be expected to function. The roles of input

and output ports are reversed, so the mirror outputs provide the 'external' inputs to the implementation, and vice versa. The roles of causal and noncausal arrows are accordingly reversed in the mirror. There is no change in liveness requirements. Required output is mirrored as required input to the mirror. The mirror insists that the implementation supply this input. This contrasts with a normal process, which may not insist upon input.

Each system event is now two process events, one output, and one linked input event. There are both causal (solid) and noncausal (dashed) arrows both to and from each system event. The causal order forms a system pomtree. The noncausal order is a subset, if the system is safe. If there is a safety violation, the system becomes undefined.

A system action is an equivalence class of system events. System events belong to the same action iff the rules for both process events are the same.

4.3 System behavior automata

Just as we did for processes, in principle we can construct a behavior automaton for a system. As before, this automaton is a finite presentation of the system pomtree. Until correctness is established, there are two relations between actions, causal and noncausal, obtained from the mirror of the specification and from the component processes

of the network.

Assuming that all the processes are honest about branching and loops, the system will also have a clean branching and recurrence structure.

Figure 18 shows a system 'molecule' consisting of two process atoms, one of an output action and the other of the input action to which it is linked. This picture can be abstracted to a 'system atom' for the system action X, that has both types of arrows on each side.

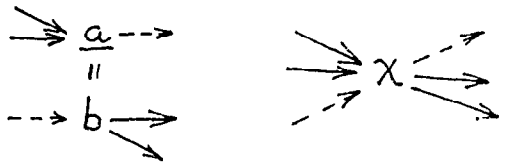


Figure 18 System molecule, atom

In practice it may be computationally expensive to actually construct a system behavior machine, and it proves to be unnecessary. Instead we lay down all the processes and their links. The verification algorithm uses these to in effect traverse the system behavior machine, visiting all the actions and checking all the noncausal arrows.

CHAPTER 5. CORRECTNESS

5.1 Definition of correctness

In this chapter we state the definition of correctness, that is, what it means for a network to correctly implement a requirements specification. We first form the closed system of mirror + network. Correctness is a predicate that holds over the "extended" system pomtree. It factors into safety (safety of all processes is respected) and liveness (the specified outputs are produced). Correctness can also be be stated in terms of language containment with respect to the causal and noncausal orders.

The chapter concludes with a proof that wires can be inserted between delay-insensitive processes without disturbing correctness.

5.2 Verification: Safety and liveness in graph terms

When a requirements specification is laid beside an open network that is a proposed implementation, one must consider separately the arrows in each.

For the specification, noncausal arrows are a promise to the implementation to supply input only as allowed by the noncausal order. Causal arrows are a requirement on the implementation to produce output only as allowed (safety), and as much as allowed (liveness). Together, this means exactly as allowed.

For (processes in) the oper network, noncausal arrows are safety requirements that must be met by the other

processes in the network, and by the environment, acting together. Each noncausal arrow must be supported by a chain of arrows, that are causal in other implementation processes, and noncausal in the specification. Causal arrows are statements of what output the process will produce. Implementation processes do not require liveness from their environment.

When the mirror of the specification is used to form a closed system, all safety is unified. The correctness question becomes, "Are all noncausal arrows supported by chains of causal arrows?" Each noncausal arrow in the "extended" system pomtree represents a safety requirement that must be supported by the causal order. Each can be verified by finding a chain of causal arrows that supports it. This includes noncausal arrows in the mirror.

For the mirror, liveness is required for input to the mirror. Input to the mirror must arrive (from the implementation) when required by mirror preconditions, that are outputs from the mirror (to the implementation). The implementation is not allowed to wait for additional preconditions.

5.3 Successor arrow checking

Arrow checking is a means of verifying safety (and optionally liveness) at states not actually visited by the

verification algorithm. It is sufficient to show that at any state at which an output event is enabled, the corresponding input event is also enabled (safe). This is the case if there exists a causal chain supporting each noncausal (safety) arrow.

Both safety and liveness can be checked in a combined way, as both involve supporting dashed arrows with chains of solid arrows. This is most conveniently done by comparing finite sets of ancestor events.

For every system event, if the input is in the mirror, both safety and liveness are checked, otherwise only safety. The noncausal preconditions of this input event are easily determined from its process definition, call it process **P**. This gives the noncausal preset of output events in **P**.

The system causal order is then searched, backwards from the output event, for output events in **P**. There are two cases. If **P** is not the mirror, we find a subset of the causal ancestor set, specifically, the set of all causally preceding output events in **P**. Earlier occurrences of the same actions are (trivially) ancestors also. Thus the search can stop when repeated events are encountered, and a multi-set is not required.

If **P** is the mirror, the search is for the causal preset. This is the set of nearest output events in **P**. Since combined safety and liveness checking is on, the search for

the preset can stop when preset members are encountered.

The correctness requirement for safety only is that the noncausal preset is a subset of the causal ancestor set. It takes at least as many inputs to produce an output as required. We call this output conservatism.

The requirement for liveness is the reverse, the causal preset must be a subset of the noncausal preset. Given delay insensitivity, liveness and safety checking can be combined. The combined requirement is that the two sets are equal.

There is a complication that can arise whenever it is possible for multiple arrows with the same label to be in a cut. This occurs with buffers, for example. It is necessary that the action found by searching is in fact the same event as the noncausal precondition. This can be determined by selecting an arbitrary cut as a reference, and then determining how many events back the causal and noncausal sources are. For liveness, they must be the same. For safety only, it is permissible for the noncausal event to be the earlier of the two.

5.4 Language containment

Because of asymmetry of control, correctness as a language containment problem is seen as follows: The system has two partial orders, causal and noncausal. The causal

order can be more restrictive. There can be stricter ordering of events. There can be less output (causal) choice, and more input choice. This means that the language induced by the causal order (solid arrows) is a subset of the language of the noncausal order (dashed arrows).

This is in contrast to the commonly asserted requirement that the language of the implementation must be contained in the language of the specification.

5.5 Wires are safe

In contrast to speed-independence assumptions, delay insensitivity treats wires seriously as processes with delays and safety requirements. The protocols (well behavedness rules) ensure that wires don't affect correctness, so our verification algorithm doesn't have to include them after all. We now prove this result for our model.

We start by considering wires as full-fledged processes. A wire has a safety requirement, that it has finished delivering its output before it receives a new input. The possible failure is called transmission interference. Wires also introduce extra causal arrows into the system pomtree.

Theorem 1. Introducing a wire into a safe delay insensitive system preserves its safety and liveness properties.

Proof. Without loss of generality, consider one point contact x = y, at one event in the system pomtree. Introduce a wire, o - - -> x ———> y between them. Figure 19 illustrates this proof. A chain of causal arrows is indicated by ==========>.

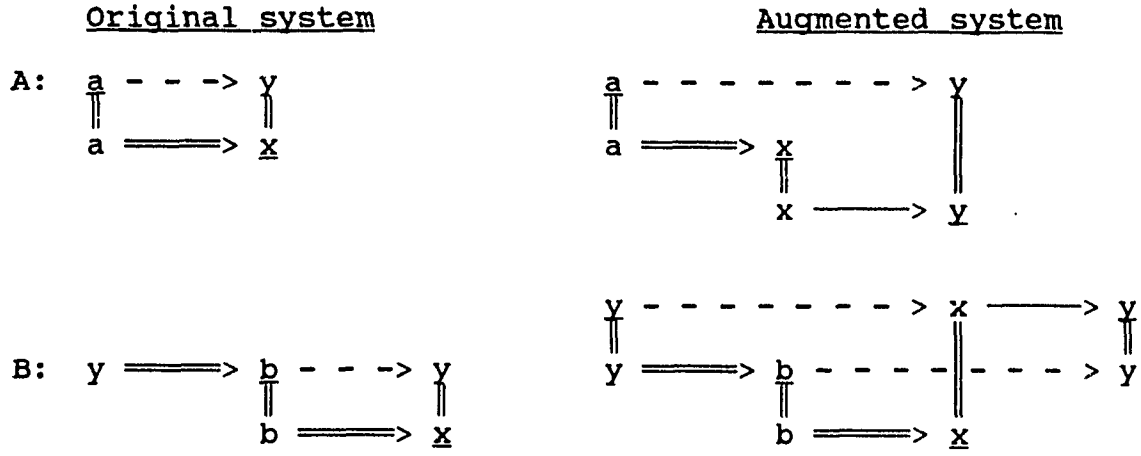


Figure 19 Wires are safe

Part A: The event y is still safe.

Again without loss of generality, consider any non-causal arrow a - - -> y. By the safety assumption, this is supported by a causal chain a ==========> x. In the new system, there is a causal chain a ==========> x=x ———> y=y, that likewise supports the causal arrow.

Part B: The wire is safe, i.e. y - - -> x is supported by a causal chain in the system.

By rules 1 & 2, there must be an output event, call it b, separating the previous occurrence of y from this y. This means y ⟹ b = b ⟹ x by the safety assumption, which supports y - - ->x.

Part C: Liveness is preserved, since the wire is live, all the augmented causal chains are as live as they were.

Theorem 2. Given a safe delay insensitive system with point contact between the various ports, the system obtained by replacing these contacts with wires is also safe.

Proof. By rule 6, there are a finite number of ports, so there are a finite number of point contacts. We can introduce the wires one at a time, by induction, all these system transformations preserve correctness by theorem 1.

CHAPTER 6. CLOSED SYSTEM EXAMPLES

This chapter gives examples of processes from chapter 3 composed into networks or used as specifications. It highlights some of the methods and problems of the verification algorithm. A one-transition buffer is implemented by gates. Many-transition buffers are implemented by chains of 1-transition buffers. Arbiters are implemented by rings of DME elements.

6.1 1-transition buffer

A 1-transition buffer has been implemented by Martin [7] using a C-element and a fork. Figure 20 is the buffer specification as given in figure 13.

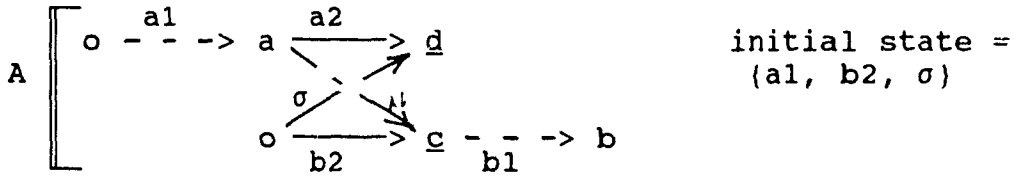


Figure 20 Behavior automaton for buffer

Figure 21 shows graphically how the implementation, consisting of a C-element and a fork, is verified by checking that all the dashed arrows are supported by chains of solid arrows.

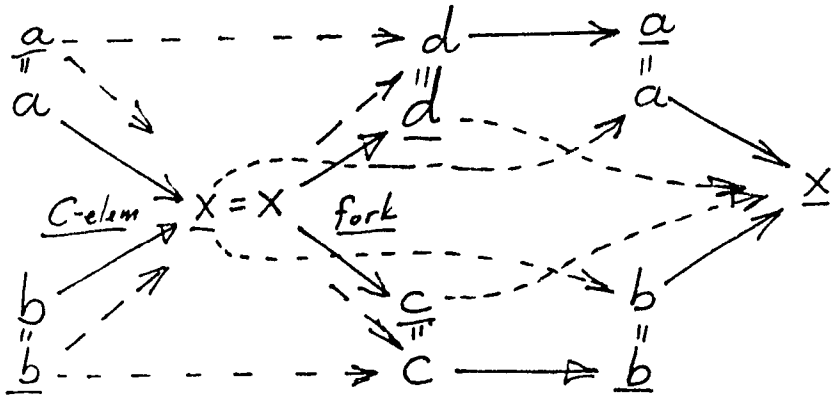


Figure 21 C-element and fork implement buffer

Figure 22 is the circuit diagram. Note the negation on one input of the C-element. This implements the initialization of one interface as active (ready to send), using the convention that all ports are initialized at logical 0. So the first transition at the other input causes the C-element to fire. This implementation may also be found as the control circuitry for a micropipeline [5].

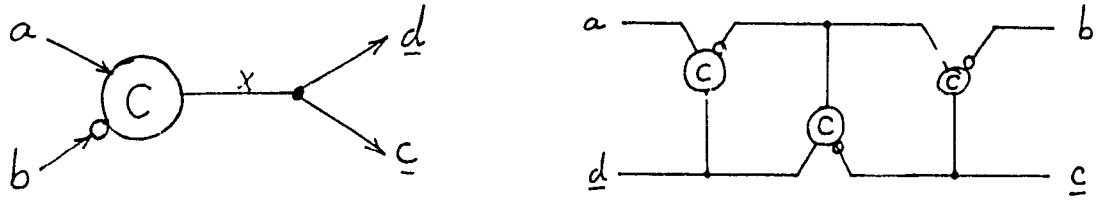


Figure 22 Circuit diagrams, 1- and 3-buffer

This 1-transition buffer may be composed to form buffers of larger sizes. The command for all sizes of buffer is exactly the same! The only difference is that the initial state contains as many σ arrows as there are places in the buffer. This is proved in the next section. Figure 22 also has the circuit diagram of three buffers which make

a 3-transition buffer.

6.2 n-Transition buffer

Figure 23 Shows graphically the proof that a m-buffer composed with an n-buffer implements an (m + n)-buffer. (shown for 1 + 2). The specification command needs to be gone through once, to verify that the events do occur. Most of the dashed arrows are verified by interface projections. That leaves only the μ and σ arrows. Each must be checked only once. μ is obvious.

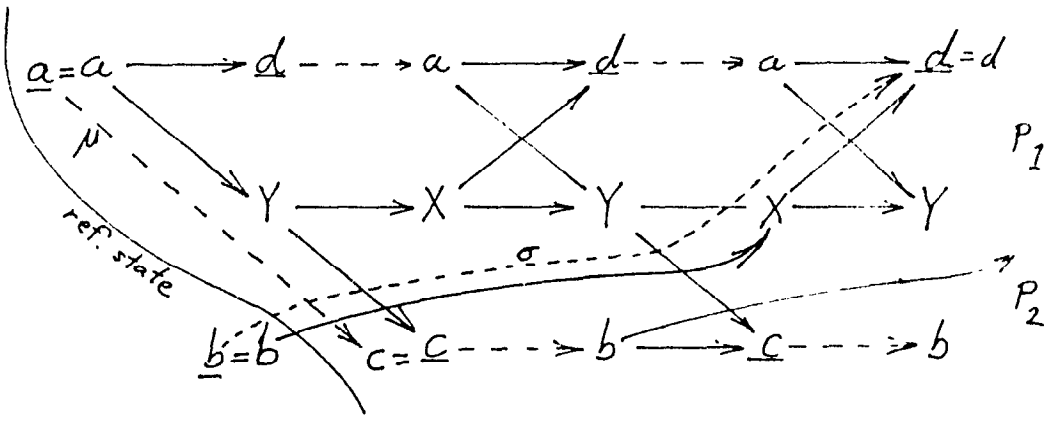


Figure 23 1-buffer + 2-buffer = 3-buffer

The last σ arrow crossing the reference state (dashed in the mirror) goes to the 3rd ((m+n)-th) d after the reference state. It is verified by the last σ arrow in P_2 going to the 2nd (n-th) X, from where the σ arrow of P_1 goes to the 3rd (m+n -th) d following the reference state, as required.

In the algorithm, we simply count the number of times

the causal chain crosses the reference state, which must agree with the number of noncausal arrows in that state. This is because each arrow name is an equivalence class of arrows. Each causal arrow in the chain that is 'in the state' stands for another arrow that is in another chain (with the same arrow labels) that crosses the state (see figure 33 in section 7.6).

6.3 Implications for matching buffers of different sizes

INFINITE BUFFER: An infinite buffer may be viewed as the limiting case of a finite buffer, with infinitely many σ arrows present at initialization. This means of course that the $\underline{d}$'s are never held up for want of a σ arrow, so the σ arrows effectively drop out. The following discussion includes infinite buffers.

EXTERNAL SAFETY: In the original form of buffer, it is clear that a larger buffer can be used to implement a smaller one. This is because the environment is enforcing safety, and will be expected to not even completely fill the buffer. The other way around, if the implementation is smaller, the environment can be expected to cause a safety violation.

INTERNAL SAFETY: In buffers of the new form, with the safety internalized, we could expect the reverse. In fact, when the implementation is a larger buffer than the

specification (let's say a 1-buffer is implemented by a 2-buffer) we will get a safety violation at the second d because the 'mirror' isn't expecting to see that d until at least one b is input. The other way around, we would observe a progress violation when the implementation wouldn't allow the 2nd d in the same situation, but no safety violation this time.

Yet there are no safety violations along the interfaces in either case, and we might expect either size buffer to be suitable in some situation. In this case we would have to conclude that the buffer is overspecified. We should perhaps include in our formalisms the possibility of variable specifications, such as "I would be happy with a buffer of capacity anywhere between 10 and 20."

Also observe that any finite buffer implements an infinite (ideal) buffer if we remove some progress requirements.

6.4 n-Bit buffer

This example has input choice. Instead of buffering simply n transitions, one could buffer actual data, the simplest being just 0's and 1's. In [5] the control structure is delay insensitive, but not the data channels.

A specification for an n-bit buffer follows the pattern previously given, each interface has three ports instead of

two: a handshake, passing a 0, and passing a 1. The behavior automaton is given in figure 24. As before the size of the buffer is determined by the number of σ arrows in the initial state.

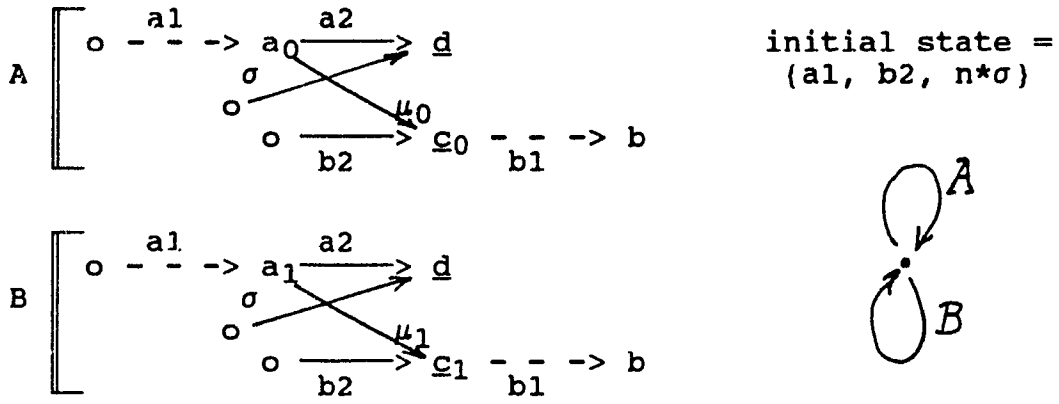


Figure 24 n-bit buffer

In this buffer, a state cut may contain several μ_0 and μ_1 arrows. They are ordered by the order in which the arrows are "produced" and "consumed". In fact these arrows are orthocurrent [2] to the lines of the interfaces. This order must be recorded as part of the state if the state information is to record the FIFO nature of the buffer, rather than just the number of each arrow. Needing to record order of some arrows is indeed a new complication, which arises from the fact that we have equivalenced many different arrows of the same type.

It is only our recording of the state that does not contain the order information, the orthocurrence is clearly reflected in the pomset, and accordingly this is not a problem for the arrow checking part of the verification

algorithm.

Figure 25 shows an example of 3 1-bit buffers, forming a 3-bit buffer. This is a straightforward generalization of the transition buffers. In the computation shown, at the first (unstable) cut a 0 and then a 1 have been input, the 0 is about to be output by P3, P2 is empty, and P1 is about to pass on the 1. Progress requires this system to progress to the second (stable) cut in absence of further input. In this state P1 and P2 are empty, and P3 holds the 1. The state of the specification does not show where the empty spaces are in relation to the data. In other words, the σ arrows are not ordered with respect to the μ_i arrows.

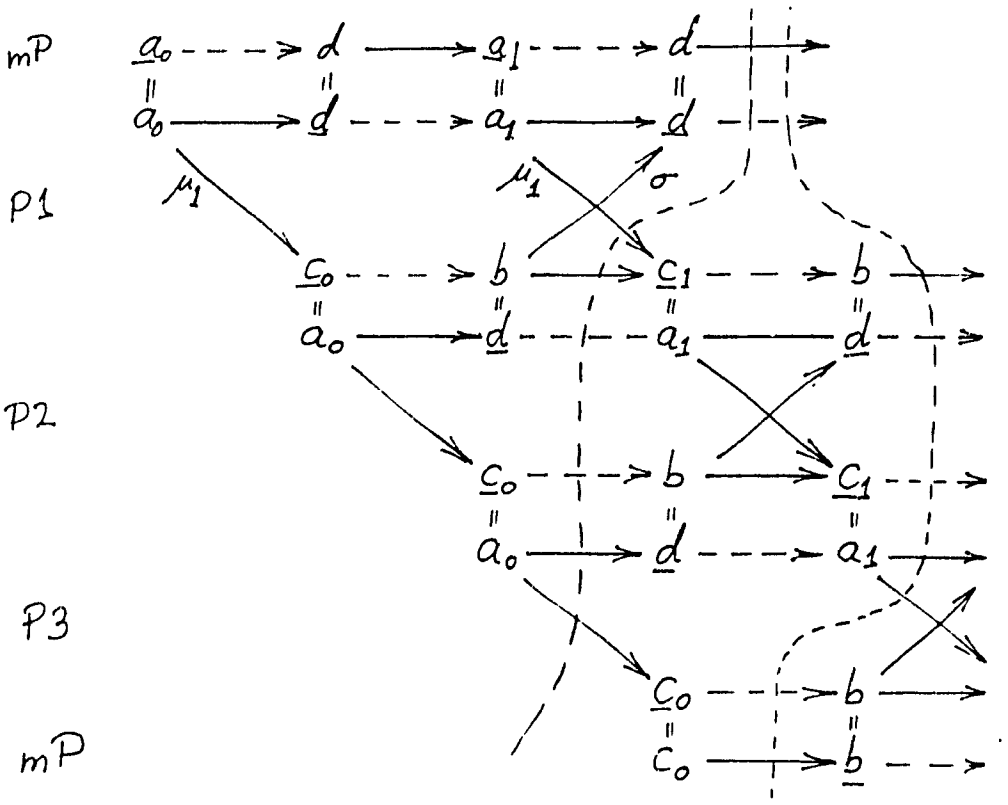


Figure 25 3 1-bit buffers

6.5 DME ring

Figure 26 shows the configuration of a ring of 4 DME elements, that should implement a 4-way arbiter. Note that each DME has a user interface, and 2 neighbor interfaces. Most of the safety checking of the system is done within these interfaces. The ring functions by passing a token in one direction in response to requests between neighbors. One of the DME elements is initialized as having the token, shown by •. Rings of other sizes are connected similarly.

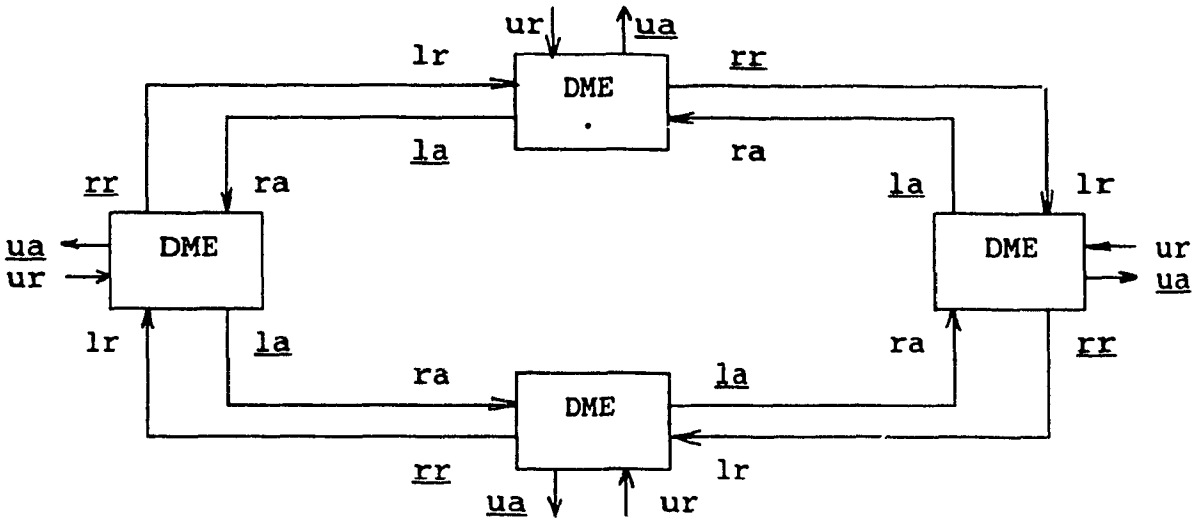


Figure 26 Ring of DME elements

Figure 27 shows the branching structure of the verification algorithm for a ring of <u>two</u> DME elements implementing an (two-way) arbiter. The line straight down corresponds to livelock, in which the token has been passed around the ring with out any events being observed externally. This appears to be a basic property of the ring structure, interesting in its own right. It also has implications for a verification algorithm, that should not loop endlessly on this possibility.

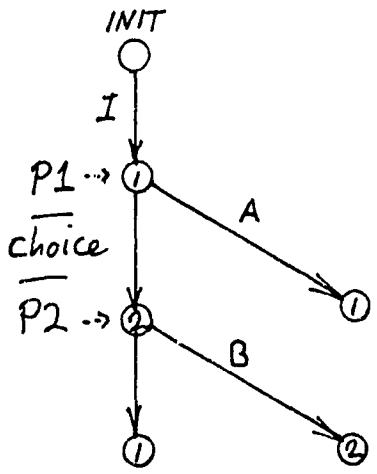


Figure 27 Branching of DME-ring verification

Figure 28 shows arrow checking of the one non-trivial dashed arrow (token in the arbiter mirror) for the same system. All other noncausal arrows have directly matching causal arrows across an interface. Checking is done backwards from a (mirror) input action, **p** in this case. Causal arrows (———>) and system links (=) are followed. The search branches (shown by large brackets,]) at DME **t** arrows, since they have alternate sources. The search terminates successfully when a mirror output action is found, indicated by circles. It terminates unsuccessfully when a repeated event is found in the chain. The leftmost events in figure 28 are interesting, as they show the possibility of the token being passed completely around the ring. The a^- is thus in a different past than the other a^- that was found. The $\underline{la}_1$ terminates the search into pasts where the token has gone

more than once around the ring without any external events. This is the potential livelock of the ring structure. The branches along 12 arrows also terminate.

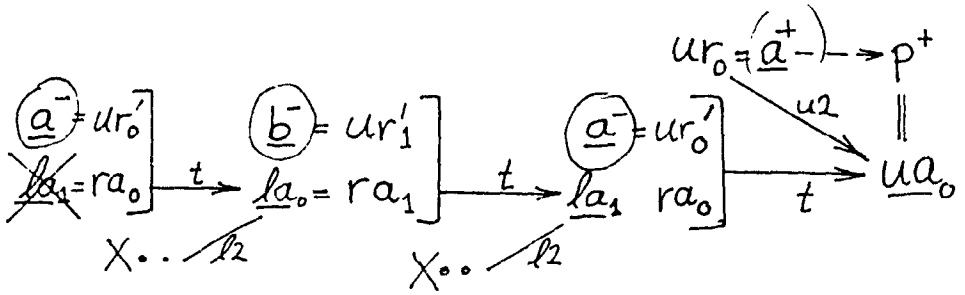


Figure 28 Checking arbiter token arrow

We have used this example as a test case for the delay-insensitive verification program, along with the more general case of a 2-out-of-n arbiter, with the ring initialized with 2 tokens.

CHAPTER 7. THE SYSTEM VERIFICATION ALGORITHM

This chapter presents our verification algorithm. It consists of two levels. The first moves through system actions, visiting them all. This is not a simulation, but a means of constructing the system actions. The second checks all the arrows leading to each action (in every context), verifying safety at states not directly visited. We establish that the algorithm terminates, either by detecting an error, or by completely covering all the actions.

7.1 Initialization

1: In process specifications, we 'mark' choice events (both causal and noncausal) and loop events (events occurring at the beginning of a recurring determinate command). Causal choice events (output) are a subset of these marked events. During verification, we will pay attention to all choice events, and loop events of the specification, in order to terminate after a finite amount of work. This does not ensure termination in the case that two or more implementation processes chatter back and forth in a determinate manner. This would put the algorithm into an infinite loop, whether there was a problem of determinate livelock or just an isolated oscillation. To guard against this possibility, optionally we can also pay attention to loop events of the implementation processes, at the expense of recording more states in our table.

2: Mirror the specification by exchanging the roles of

In and Out ports. This changes causal arrows to noncausal, and vice versa, giving an environment that drives the implementation. Construct the set of 'commands'. Each command is a set of actions that start from marked events, and lead up to the next marked event(s). Completeness of each command is a liveness requirement.

7.2 Visiting system actions, checkpoints

3: We effectively construct the finite behavior automaton of the closed system **S** by moving through (visiting) all the actions in the system that are reachable from the initial state, by recursively calculating the causally enabled actions, and visiting them. For each action visited, we check arrows and update the state. It is important to note that only a fraction of the possible states are visited. Since this is not a simulation, visiting can be done in any arbitrary order. However, we defer visiting any 'marked' action as long as there are unmarked actions enabled. Then the current state is recorded in a table of checkpoints. The purpose of this strategy is to come back to the same state again, ruling out the possibility of moving past it non-sequentially. This keeps the termination table small. If the state has been reached before, the recursion terminates. If any of the enabled actions are causal choice actions, the recursion 'forks', visiting several possible

futures.

These specially constructed and recorded states of **S** are now the loop or choice states of the finite behavior automaton of **S** (nodes of its dsfm), and the moving algorithm stays entirely on one command while moving between these states. Upon termination, all reachable actions in **S** have been visited, and all commands of the behavior automaton of **S** have been traversed. In other words, we have traversed all the sticks of the system stick figure.

Since we have marked all the initial actions of the commands of the specification **P**, we are always on zero or one command of **P**. A command is sometimes loaded, and sometimes not. For example, no command will be loaded if a choice is made to perform an internal action. As well, all the nodes of **P** are present in the recorded states of **S** because of our convention for picking them. We minimize the number of states of **S** that are recorded by choosing to visit actions in a regular and consistent manner, visiting all internal, non-choice actions, and all external actions that are not at the start of a **P** command before recording the state in **S** and taking a checkpoint.

For example, Figure 29 (a) gives the stick figure for a 3-way arbiter **P**, while Figure 29 (b) gives the stick figure for system **S** implementing it by a ring of DME elements. The three recorded states of **S** all correspond to

the choice state of **P**. They also represent the possible locations of the 'token'. The sticks between them correspond to (internal) token passing. No **P** command is loaded while on these sticks. The loops correspond to the commands of **P**.

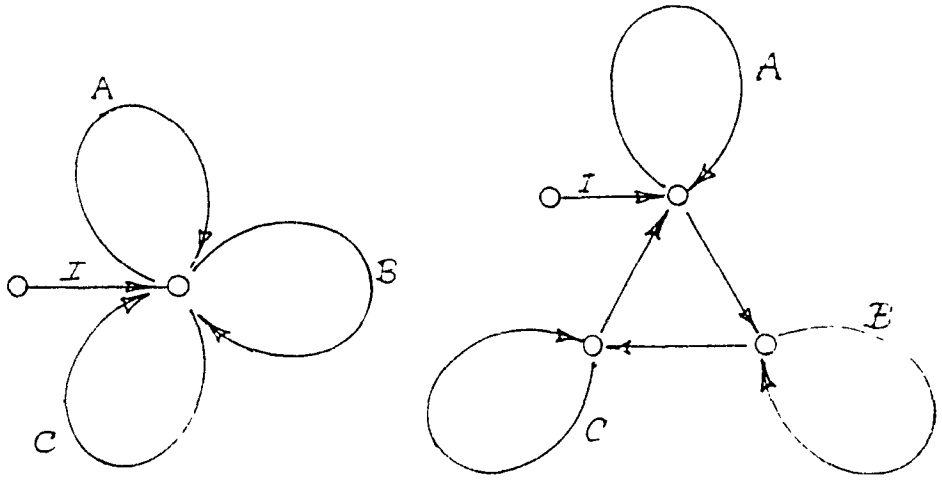


Figure 29 Stick figures (a) arbiter, (b) system

There is elementary safety checking on this level. If a system action (the output half) is causally enabled but noncausally forbidden (the input half), we have stumbled upon a safety violation. In the determinate case, the arrow checking would also pick this up. In the non-determinate case, however, we rely on this check to rule out any possible unsafe occurrence where output selection is involved.

7.3 Liveness

4: Liveness requirements of the specification are verified. There are two issues. First, all the required output actions must be produced. These are linked to inputs of the mirror, or implementation tester. (By contrast, implementation processes cannot demand input). As we visit actions starting from each checkpoint, we load the command containing the first specification action. The moving algorithm attempts to complete this command once it is started. It is sufficient liveness checking on this level to require the completion of commands, and stop with a liveness violation otherwise. The only other condition, when no command at all is started, is easily detected as total deadlock.

Secondly, when an specification output (mirror input) action occurs, check that no additional inputs from the specification are required to produce a given output. This is done by comparing causal and noncausal presets. A causal preset may not contain specification actions in addition to those required in the corresponding noncausal preset. This motivates the backward searching described in the following section.

7.4 Arrow checking for combined safety and liveness

5:(a) While visiting each system action (linked out-in pair), if the In action is in the mirror of the specifi-

cation, noncausal arrows are checked as follows, for both safety and liveness. (In part 5:(b) below, we modify this for checking safety only when the In action belongs to some implementation process P_i.)

This action represents the equivalence class of all like named events in the pomtree of the specification. Using the set of atom rules of the specification, determine the sources of all the arrows needed by this action. In the absence of backwards conflict, this gives a single (noncausal) preset. In the presence of backwards conflict, it gives a set of alternate presets, because different events may have different histories of events preceding them in the partial order of the pomtree. Thus, in general, we obtain a set of noncausal presets.

Likewise, the action pair represents the class of all like-labeled system events. Using the sets of atom rules of all processes, and their links, search backwards along causal arrows, until specification events are reached. Searching can be abandoned whenever an action is repeated in the causal chain. This guarantees a finite search.

Such a termination condition is particularly important for non-determinate processes, as it factors out action loops in actual chains between events. For example, in a ring of DME elements, the DME token can be passed repeatedly around the ring before a user request is granted. This

results in a cycle in the chain of causal arrows supporting the arbiter's critical section arrow. Such a cycle must be factored out, as is done by this procedure. As far as the algorithm is concerned, the important thing is to avoid an infinite loop during verification. As far as the example is concerned, this is non-determinate livelock, which requires an appeal to fairness of choice to ensure progress.

When any causal arrow has alternate sources, this represents a backwards conflict, so alternate presets are calculated. The search is recursive. Upon return from the recursion, alternate sources of the same arrow return alternate presets, while multiple arrows needed by an action result in unions of the presets of each arrow. This becomes a cartesian product if there already are alternate presets.

For example, suppose x needs two arrows. One arrow has the preset {a}, the other has alternates {{b}, {c}}. The presets of x become {{a,b}, {a,c}}. See figure 30. Wavy arrows represent chains of causal successor arrows, and # indicates a backwards conflict.

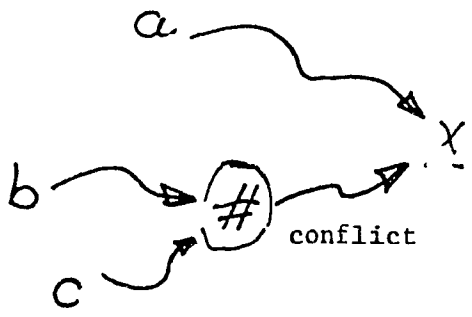


Figure 30 Causal presets

Finally, the causal preset(s) is (are) compared with the noncausal preset(s). As processes are allowed to be input liberal, there may be fewer noncausal presets. Each noncausal preset must exactly match some causal one. Too few causal preconditions would be a safety violation; too many would be a liveness violation. In fact, the presets may be incommeasurate. Additional events in the causal preset are liveness violations. Missing events may be safety violations, or if they are ancestors, may not be direct preconditions as a result of some other violation.

Figure 31 is an example of an input-liberal implementation that has more non-determinism than the specification. The specification is an alternating merge, and the implementation is a general merge.

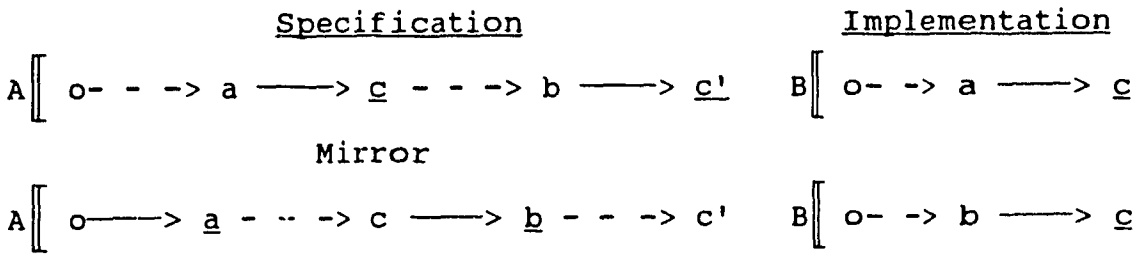


Figure 31 Input-liberal implementation

The noncausal preset of c' is {b}, since the specification is determinate. Its causal presets (from the implementation) are {{a}, {b}}. The noncausal preset equals one of these namely {b}.

Note that the search for causal preconditions only looks for causal chains without intermediate specification events. This actually makes the combined safety and liveness checking of specification events shorter than just safety checking, discussed next.

7.5 Arrow checking for safety only

5:(b) When the input action belongs to some implementation process P_i, we check for safety only. This is done by relaxing the above arrow checking in two ways. First, the search continues past events of P_i, stopping only on repeated events, (since liveness is not required, additional events may intervene), second, every noncausal preset must be the subset of some causal ancestor set.

For example, suppose the implementation is 2 wires, but in the specification they are used alternately, as in figure 32:

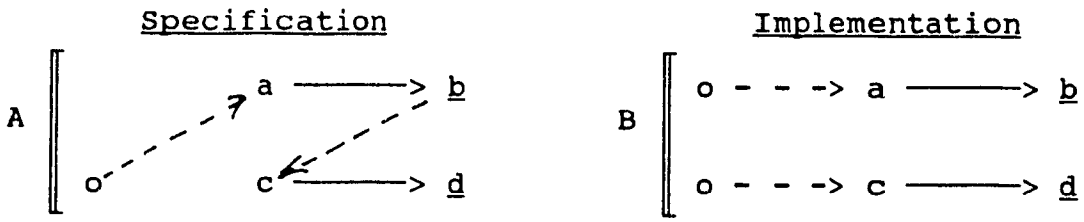


Figure 32 Two wires used alternately

The noncausal preset of a is {b}, but the causal ancestor set is {b,d} where in fact d follows b before a occurs. (The input event c is also an ancestor, but for delay-insensitive systems, we don't need to record input

events, since they are of the wrong type to ever be in a noncausal preset.)

7.6 Multiple arrows in a state

It is possible to have more than one arrow of the same name crossing an arbitrary cut. This happens in producer-consumer situations such as buffers. The important concept here is that each named arrow - and each causal chain - is an equivalence class of like arrows in the pomtree. (Please note that this is a different sort of equivalence than equivalencing arrows in behavior automata.) Thus whenever several noncausal arrows cross the current state (just an arbitrary reference), the corresponding causal chain must have the same number of arrows crossing the cut: these are "class representatives" for the other chains necessary to support the other noncausal arrows. The count of state crossings has to match for combined safety and liveness. For safety only, the count of causal crossings must be less than or equal to the noncausal count. For example, a one-place buffer safely implements a two-place buffer.

For example, consider figure 33, which is abstracted from a 3-transition buffer (see section 6.2). The dashed cut is the current state, the dotted cuts are earlier occurrences of the same state. Since the darker arrow b ———> d is being checked, both the noncausal arrow and the support-

ing causal chain must come out to the same b. We identify the square boxes where the current state is cut, which stand for the circles where the arrow is cut by equivalent states, indicating the third b back from the current state.

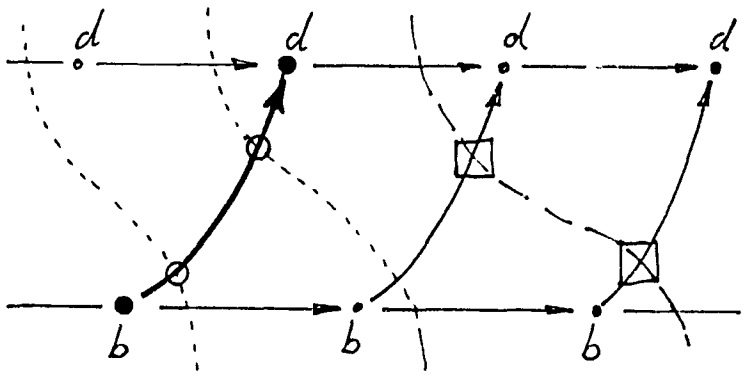


Figure 33 Multiple arrows in a state

The situation is not so clear when there is backward branching. Presets must agree, as before, but the causal count for an arrow must agree for some preset, and be less than or equal in all presets.

7.7 Completeness upon termination

Verification starts at the initial state, i.e., the state immediately following the reset event. From there the algorithm visits system events, which are event-pairs, one Out connected to one In. Each event thus has a double action label. States of **S** that meet the criteria for checkpointing are recorded in a table (see section 7.2). These checkpointed states become nodes of the constructed system

stick figure. When the verification terminates, all 'sticks' of the system behavior automaton have been traversed, and all possible action labels have been visited, since every checkpointed state that can be reached directly or indirectly from the initial state has been reached, and every action that can be reached from a checkpointed state is part of some 'stick.'

If the nodes of the system stick figure have been constructed using execution state encoding, and we terminate when the same execution state has been reached, we must establish that all its possibly different preconditions have been seen.

In other words, when the verification terminates successfully, have all action pairs that there are in the system pomtree been seen? Naturally, each event has a different history. We have actually visited one instance of the 'execution' equivalence class given by event and arrow naming. This corresponds to a finite number of distinct equivalence classes when the part of the history relevant to arrow checking is taken into account. Each member of such a 'behavioral' class is identical with respect to the arrow checking.

We next assert that the arrow checking of each 'execution' equivalence class covers all the 'behavior' equivalence classes. In particular, all noncausal arrows in

the pomtree have been safety checked, and all specification output (mirror input) events have been liveness checked.

Consider a pomtree, rooted at INIT. There are some arrows leading from INIT, corresponding to events that are initially enabled. Noncausal arrows from INIT are trivially safe, since all causality proceeds from INIT. We could equally well leave these arrows in the picture or erase them. (We choose to leave them in, to avoid extra artificial distinctions of execution states.)

The arrow checking is grammar based, rather than computation based, so it is not necessary to consider all possible execution sequences. Causal preconditions are calculated using the sets of atom rules of all the processes in the system, and the system links, while noncausal preconditions are based on the set of atom rules of the one process to which the input action belongs.

Without loss of generality, consider any one action-labeled class. The arrow checking algorithm explores all possible noncausal arrows, and all possible causal chains leading up to any of these event-pairs, by using the fixed sets of atom rules of all the processes in the system only, without any reference to a particular history. The rules for the process P_i of the input action give all possible sources of the noncausal (- - ->) arrows. Likewise, the sets of atom rules of all the processes together with the

system links establish all possible causal preconditions in P_i resulting from chains of causal arrows, including chains that have cycles that repeat (c.f. DME ring).

Liveness requires that events are actually produced, and that when produced they are the result only of the required inputs. We ensure that they are produced by insisting that P-commands, once started, are finished. If fairness is assumed, in addition all P-commands must be present in the system.

Safety and Liveness for an event-pair for which the input event belongs to **mP** is established as follows. (One of) the noncausal preset(s) of this (specification) event must be the preset for this event's history. We have found an exactly matching causal preset. This shows that the production of the event is live, that is, it is caused directly by these prior events without the necessity of any others. It also shows that it is safe, since all (non-causally) required events are needed to (causally) produce it.

Safety (only) for any other event-pair is shown similarly. (One of) the causal preset(s) must be the preset for this event's history. We have found a causal preset that is a superset of it, ensuring safety.

You may have noticed the possibility of additional causal presets. This is not a problem, it is just a consequence

of the fact that implementation processes may be input-liberal. In other words, an event may have a causal preset that is not exercised, that is, it would be willing to produce its output in response to input that is not provided by the environment. But suppose that such a 'ghost' preset did occur and lead to the causal enabling of an event. This would imply an execution state at which the event was causally enabled but noncausally forbidden, an occurrence that would be caught as a safety violation while 'visiting'.

Therefore both safety and liveness hold over the entire pomtree of **S**. The visiting algorithm generates the behavior automaton of **S**, that in turn generates the pomtree. Since it has visited one event representative of each action and invoked the arrow checking algorithm, which uses only the sets of atom rules of the processes mP and P_i, anything that might appear in the **S** pomtree is checked.

7.8 It does terminate

As we move through events, we assume that after a finite number of events, the only events left enabled are 'marked' ones. (This can be assured by marking the beginnings of all commands.) Whenever this happens, we record the state of **S**. We can visualize this as putting a dot on a state of the finite state machine of **S**. We stop whenever we reach a state with a dot on it. Otherwise, we proceed,

perhaps with finite branching in case of choice. As there are only a finite number of states in **S**, the algorithm must terminate in finite time.

Of course, by arranging to place dots only at states corresponding to starts of commands of P, and finishing off internal events in a regular manner, we expect to finish after placing only a small number of dots.

7.9 Summary

The network verification algorithm has been presented. There are two levels. All the system actions are visited by starting at the initial state and proceeding based upon events enabled at the current state. At all actions, arrow checking ensures safety (and liveness for mP).

Comprehensiveness is assured since all system actions are enumerated, and all backing up from an action for arrow checking is done without any reference to the execution history that led to the action, but is based entirely upon the sets of atom rules of the processes and the system links, both of which are fixed.

Finite termination is assured under the assumption that only a finite number of events are visited between recording of states. This can always be arranged.

CHAPTER 8. THE POM PROGRAM

8.1 Trilogy language features

A computer program implementing this algorithm has been written in Trilogy [10,11]. Trilogy is a multi-paradigm language that incorporates procedural, logical and data-base elements. The program is written primarily in the procedural mode, using the logical and database capabilities where appropriate. For example, a data base of processes is maintained, and operations such as set union can be easily expressed in predicate (logical) mode.

The Trilogy version (1.3) used has two major drawbacks: Data is limited to a single 64k segment, and garbage collection is limited to heap cuts in certain circumstances. Some programming choices were made to enable the heap cuts. In other cases, fixed size arrays were chosen to avoid the accumulation of garbage.

As a result of these two limitations, the program has to be considered a prototype. Even so, it can handle many more DME elements than Dill's [8] state-based program, which was programmed in Lisp on a machine with virtual memory.

In Trilogy, procedure parameters have one of several modes, input parameters must have a value which cannot be changed, output parameters must obtain a value, while io parameters are like Pascal variables whose values can change, a significant restriction is they must be first initialized, all at once in case of an array. In this

program, most parameters are either input or output. The current state, by contrast, is maintained as an io array. This specification of modes of parameters is a significant help in program development.

An example of a procedure declaration, with various parameters: (4)

```
proc Sum (numbers:<IntArray, count:.I,   sum:>I) iff
          {input array       io integer  output integer}
```

The program is structured in several modules. In Trilogy, modules are compiled separately, and their dependencies are catalogued in a library. The first two modules contain type definitions.

8.2 Data structures

Processes have two representations in the program, first an external one, where ports and arrow names are strings, this is in module **process_t**. The second, internal, representation, which is defined in module **netverify_t**, uses instead scalar types and fixed length arrays instead of lists, for speed and compactness. In either case, a process consists of its name, a set of ports, each designated as

---

(4). Trilogy uses capitalized identifiers for constants, types and procedure names. Variables and keywords start with lower-case. So Sum is this particular procedure, sum is a variable.

either In or Out, the names of all the arrows, the initial state, expressed as how many arrows of each type are in the state, and the rules for events. Each rule consists of: the port, a prime distinguishing it from other rules for the same port, a mark specifying if this is the start of a choice or looping command, and lists of arrow needs, gives and tickles.

Finally, a process may have coupled choice, this must be explicitly given as sets of choice events that may be chosen together. (This capability has been implemented in the program, but as no actual examples of this type of non-atomic choice have been encountered, the data entry module **enter_p** lacks some code to allow easy entry of coupled choice.)

Systems of processes are also defined in module **netverify_t.** A system consists of one specification process, which is mirrored to drive the system, and a number of implementation processes (using the internal representation), links from output ports to input ports, and the set of commands of the specification. Each command is a list of actions starting from a designated choice or loop point (until the next one).

System state (io variable) is a vector of the states of the individual processes. Its initial value is taken from the initial states of the individual processes.

An Event in the system context is a rule and its process, an EventPair is both an output Event and the input Event to which it is linked.

A stack of choices to be explored is maintained for non-determinate systems. Each entry contains the current state and a list of events (normally choice events) to be done on this branch of the verification.

8.3 Modules

There are three 'top level' modules: enterp, setup, and netverify. They communicate by means of database files.

Module **enterp** allows process definitions to be entered from the keyboard into the database file ProcessF. There is also a procedure to generate an arbiter process of specified size, and procedures to make a process that differs from another only in its initial state, and to maintain the file by revising and deleting processes.

Module **setup** 'sets up' a system of previously defined processes. It asks for the names of all the processes and converts their definitions into internal form. This is done by proc FixForm, which in particular calls pred ConvertRule, which (optionally) unmarks loop events of the implementation processes, in order to minimize the number of recorded states. For each output port, proc AskLinks asks for the (process, port) linked input. One is asked to pick from the

list of not yet linked input ports of the designated process, unless only one remains (or none remains, which would be an error).

There are special setups for various sized rings of DME elements and buffers, for convenience, since I tested so many of these.

Module **netverify** is the main routine for verifying. NetVerify sets up an empty stack of choices to be checked, an empty table of choice and loop states visited, and the initial state. Verify moves through all the plain events, until only marked events are enabled. At this point, Hash records the current global state in the table. (5) If the state is already in the table, this branch is finished. If not, all choices are put on the stack by Fork. Commands of the specification are loaded as new branches are verified.

It is a liveness violation to reach another marked state without finishing the current command. Hash stops with a message in this case. If some specification command was never chosen, this might be considered a fairness violation. The present implementation does not flag this occurrence, which can be noticed from the program output.

Subroutine DoForks takes choices off the stack and verifies them (which can result in more choices on the stack). The whole verification terminates successfully when

---

(5). It has not proven necessary to actually use hashing techniques.

the stack becomes empty.

Procedures Move and MovePlain actually move through the enabled events, updating the state and calling ArrowCheck.

Module **enabled** makes a list of all the events that are enabled at the current state, and also contains procedures to update the state when events 'occur'.

8.4 Checking successor arrows

Module **arrowcheck** does the arrow checking for safety and liveness. This is the heart of this method. There have been several versions of this module. A very efficient version in June 1990 checked safety only, one arrow at a time.

The determinate case is quite straightforward, and can make extensive use of Trilogy's predicate mode for searching the graph. This is the case when the processes themselves are determinate, or when behavior states are recorded, so that (enough of) the past is determined. (See discussion, section 9.1).

In the most general case, nondeterminism is visible (as a backwards conflict, since we are searching backwards), and there is the possibility that several arrows can cross a state boundary. The following discussion applies to this general case. Most arrows can be checked more simply, so checking is speeded up when the procedures Safe and Direct_c

succeed in matching a single preset.

Procedure Presets determines the noncausal preset(s) of an event. Procedure Search then calls Causal to perform a depth first search (backwards) of the causal system graph to find the preset(s) of events in the same process. It backs up over an arrow either Uniquely or by predicate Pre, which finds non-determinate sources. It then calls on procedure Depth to search further backwards, for each different source. The backwards search is terminated in two ways. If liveness checking is on, it goes no farther than a specification event. It also stops short of passing an event whose action label has already been seen.

Depth may call Causal again, of course, and each alternate source gives a different preset. In fact, lists of presets are concatenated.

Causal is tail-recursive (Trilogy's answer to loops) for the breadth part of the search: Back over another arrow. It combines lists of presets by Blend, which gives a cartesian product of the lists. For example, if the source of one arrow is x, and the source of another is a or b, then the two presets are (a,x) or (b,x).

As the search is being performed, a count is kept of any 'active' arrows, that is arrows whose name appears in the current state. This is evidence that another instance of the chain being formed cuts across the current state.

This count will be compared with the status of the noncausal arrow.

Procedures Safety and Live, respectively, compare the noncausal and causal presets. As previously discussed, each noncausal preset must agree in actions with some causal preset. In the case of Live for the specification, the actions must be equal. The counts of active arrows must also be equal, except that in the non-determinate case, it is sufficient to find the required number of active arrows in another causal preset. For safety only, the requirement is that each noncausal preset be a subset of some causal preset, and each causal active-count be less than or equal to the noncausal. (E.g., a 1-Buffer safely implements a 2-Buffer.)

CHAPTER 9. RESULTS AND DISCUSSION

This program (over several versions) has been tested with simple examples, networks of DME elements, and buffers of various sizes. Results of using behavior-state vs. execution-state encodings are given. We obtained emperical estimates of time and space complexity, which is found to depend on the amount of branching and recurrence in the closed system, on the number of nodes in the system stick figure, and on the shape of the search trees during non-trivial arrow checking. Visible choice is important for keeping the number of nodes small.

9.1 Behavior vs. execution states

In one experiment, verification of the ring of DME elements was run using behavior states in the termination table. This approach has the nice feature of factoring out the non-determinism, so that arrow checking is presented with a finite pomset segment. There are two problems with this, however. The first is duplication of effort. For a ring of two DME's, there were 10 behavior states recorded in the table, in contrast to 2 execution states. Many actions were simply verified repeatedly.

The second problem is that it was not immediately clear just how much historical information needs to be retained, and there must be a limit to insure a finite (and reasonable!) number of behavior states. It is necessary to know

the sources of arrows, but more than just the arrows in the current state. Consider a noncausal arrow being checked. It must be supported by a chain of causal arrows, of which usually only one will be in the current state. Thus as a first approximation I retained the most recent source of all the arrows.

For the DME ring this was not enough, because it has the nasty property that the token can be passed completely around the ring without causing any external events. When this happens, the token arrow of the arbiter records the last user that gave back the token, but arrow sources in all DME elements only record that they have received a token from the next DME element. The program actually reported a safety violation at this point.

To handle this complication, the program would have to recognize that the present behavior state is equal as an execution state to one previously recorded, and realize that the transition to the new state has occurred without any external actions. We could then formulate an argument for 'factoring out' this internal loop. In view of the first problem, I did not pursue this option any further.

Execution state recording, on the other hand, leaves the non-determinism visible to the arrow checking procedure. Since the search of the graph is backwards, what must be checked are the arrows that have alternate sources,

representing backwards conflict. In the case of the arbiter, these are the same as the choice arrows, but this is not necessarily the case. For example, a coin-flipping element (i.e. a ———> b/c - - -> a) has separate forwards and backwards conflict arrows, as does a merge element.

The (2-)Arbiter's critical section arrow can be thought of as 4 possible arrows to be checked. The two destination actions are chosen by the top level verification procedure. For each of these, the arrow checking procedure looks back at the two possible sources, thus verifying the arrow in its 4 distinct instantiations. We therefore make the claim that when the algorithm terminates, representatives of all these arrows appearing in the system pomtree have been verified.

This is not quite true, since there are arrows from initialization. Initialization can be thought of as a pseudo-event at the beginning of time. There is no need to verify noncausal arrows from initialization, because all causality follows initialization.

9.2 Performance on benchmarks

The program has been tested with rings of DME elements and buffers. Space complexity was found to be linear in number of processes, except for the termination table, whose size is the product of arrows in a checkpoint times number of checkpoints. This size is linear for buffers and

quadratic for DME rings. Time complexity depends essentially on the structure of the system behavior automaton.

We obtained the execution times for somewhat optimized versions of the program, using a Commodore PC 10-I (compatible with an IBM PC) with an 8086 processor running at 7.5 mHz. A RAM-disk was used for the 'file' StackF holding the stack of deferred branches. (6) Thus, the only physical disk access is to fetch the initial setup. Extra disk accesses would increase the times, and their variability due to head movement.

Table 1 gives the time required (in seconds) for rings of DME's. N is the number of DME's, and also the number of states recorded (one for each possible location of the token). These times were obtained with a version that does not allow multiple arrows in a state. The general version went as far as n = 9, with a time of 192 seconds.

Table 2 gives the timings for an n-Buffer implemented by n single buffers. There is always just one table entry here, since the system is determinate. A relatively large amount of the space and time is taken by arrow checking for the chains of arrows that extend between the interfaces. Even so, the time requirements are linear for buffers. The version of arrow checking that produced table 2 was opti-

---

(6). The library distributed with the program catalogs this file in the default drive and directory; you can change this if you have a RAM disk installed.

mized as follows: The search for causal preconditions has been further pruned to stop not only at repeated actions in the same causal chain, but at any action previously seen in the search. This is reasonable for the determinate case. Unfortunately, it is neither appropriate or correct for the non-determinate case.

Both tables have been taken to the limit of Trilogy 1.3's 64k data segment. In each series, the next larger network gave a heap overflow message.

TABLE 1

Timing for n-Arbiter implemented by DME ring (n states in table)

=n=	Seconds
2	19.11
3	30.43
4	44.43
5	61.46
6	81.78
7	105.57
8	133.09
9	164.50
10	200.15

TABLE 2

Timing for n-Buffer implemented by single buffers (always 1 state in table)

=n=	Seconds
2	6.32
3	8.08
4	9.83
5	11.43
6	13.13
7	14.89
8	16.59
9	18.35
10	20.05
11	21.92
12	23.67
13	25.49
14	27.30
15	28.84
16	30.70
17	32.57
18	34.44
19	36.37
20	38.28
21	39.21
22	41.36
23	43.39

Polynomials are fitted to both of these sets of data using the least-squares method [12]. Tables 3 and 4 summarize the results. The buffer series is clearly linear. There is one command, in which internal events increase with n. The chains of solid arrows checking the non-trivial dashed arrows also grow proportionally to n.

Space requirements for the buffer series are also linear, for all of the following items: more processes, more space to store system state, size of the one entry in the termination table, and working storage, particularly during arrow checking.

TABLE 3

Curve fitting for Buffer elements

order	c_0	c_1*n	c_2*n^2	standard_error
1	[2.607865613,	1.763043478]		0.19072
2	[2.879771316,	1.704443111,	0.002344014681]	0.17348

The arbiter series has linear space requirements except for the termination table, which is $8n^2 + 4n$ bytes, since there are more arrows in each state (from the larger number of DME elements), and n states in the table. Hence the termination table size dominates for large n. We held the termination table constant for the series, and observed total space requirements of 35K + 2.5K*n.

Analysis of the algorithm leads us to expect quadratic time for the arbiter series. There are n arbiter commands,

and n token passing branches. Each command checks one token arrow, with time roughly proportional to n. The exact proportionality depends very much on the shape of the causal search tree, this one grows taller and remains skinny. This arrow check takes a large portion of the time. Table 4 gives the curve fitting:

TABLE 4

Polynomial curve fitting for Arbiter

order	c_0	c_1*n	c_2*n^2	c_3*n^3	c_4*n^4	std_error
2	[9.675238,	1.573463,	1.740822]			0.603795
3	[3.991904,	5.338867,	1.036277,	0.039141]		0.025779
4	[3.713333,	5.591464,	0.960682,	0.048232,	-3.79e-4]	0.024893

The Arbiter series is fitted adequately by a quadratic, while increasing the model order to 3 gives an improvement in the standard error measurement. An F-test [13] indicates that going to order 3 is statistically significant at the 1% level. Order 4 adds nothing.

The unexpected appearance of an n^3 term in the program times requires explanation, which is as follows. At several places in the program, we make use of Trilogy's logic capabilities. Essentially Trilogy uses backtracking to produce a list of solutions satisfying a predicate. This list is "sorted in a standard order, with duplicates removed." Now, if this is done n times, and the list is proportional to n items long, and the sort is $O(n^2)$, we have some $O(n^3)$ activity. Order doesn't matter to us. When we want the set

of enabled events, any way of handing it over would do. When we compare presets, a more efficient sort could be used, or another set representation that would not require sorting for set comparisons.

For example, I removed just one of these unnecessary sorts, and the n^3 term dropped from 0.049 to 0.039. The time for n=10 dropped by 20 seconds.

We conclude that the essential performance of the algorithm for this benchmark is quadratic in both space and time.

9.3 Importance of visible choice

Performance of the algorithm is very much dependent on the particular closed system produced by coupling mirror and implementation. This in turn is a function of the choice of process specifications. Specifying choice only when it is visible is important to efficient verification.

As an experiment, an alternate version of the DME element specification was used which followed the hardware implementation. It decides, in advance of obtaining a token, to which interface it will grant a request. This leads to a hidden choice, a choice made early, when the only visible action of the process is to request a token from its neighbor DME element.

The result was a program run with many more than n

states in the termination table, and correspondingly worse execution times. It is not hard to see why. At every choice state for the arbiter, the state of every DME element is reflecting a hidden choice that it has made. In fact there are n concurrent independent choices to deal with, instead of just one at a time by the DME element with the token.

9.4 Other tests

Other verifications that have been done include the implementation of a 1-Buffer by C-element and fork, a fork-join process, and a 2-out-of-n arbiter for n = 4 and 5, implemented by a ring of DME's. In the latter, the only change is that the initialization of the arbiter has 2 critical section arrows (called 'token'), while 2 of the DME's are initialized with a token. The arrow checking verifies that the proper number of dashed 'token' arrows are supported.

Some attempted verifications found bugs in proposed implementations. Verification stops when the first bug is found. Such tests include implementing buffers with the wrong size buffer, having too many tokens in a DME ring, and trying to match up 2 wires with an arbiter.

Appendix 2 has sample output from some of these runs.

CHAPTER 10. CONCLUSIONS

10.1 Summary

It is possible to represent processes using partial orders, and pomtrees in particular. States of a process are related to cuts through the pomtree. This forms the basis of a practical network verification algorithm. Delay-insensitivity rules simplify the algorithm. Safety and liveness properties of a system are defined in terms of the graph of the system pomtree. This approach has been shown to avoid the 'state explosion problem' associated with modeling concurrent systems as finite state machines.

The state explosion problem arises in more traditional approaches because one evaluates some state predicate at each state of the concurrent system, and the number of states is related exponentially to the number of concurrent actions. In these approaches, more concurrency means more states and more work to do. In our approach, we compare noncausal and causal orders, by checking the transitively-reduced arrows. More concurrency means fewer arrows and accordingly less work to do.

Successor arrows can be named, and these names form the basis for encoding process states, for recording states in a table of checkpoints, when necessary, and for polynomial time system verification. We use arrow names in the encoding of behavior automata, which are finite presentations of processes that have a clean branching and recurrence

structure.

We gave well-behavedness rules that are appropriate for hardware processes in general, and delay-insensitive processes in particular. The main requirement for use of the algorithm presented in this thesis is that safety and liveness of a process can be expressed in terms of causality, that is to say, by specifying what events are enabled (or required) on the basis of preceding events.

To verify that a network of processes implements a requirements specification, we form a closed system by connecting the network to the mirror of the specification. The mirror acts as an implementation tester, providing the worst possible environment for the proposed implementation. In this closed system, all the safety requirements, both of the mirror and the processes of the implementation, unify into the requirement that all noncausal arrows are supported by chains of causal arrows.

The verification algorithm that has been developed contains essentially two levels. The top level moves through system events, branching on choice, until all the different system actions that can be reached from the initial state have been visited. The second level, arrow checking, actually verifies safety and liveness over the system pomtree by comparing the noncausal and causal partial orders leading to each event. This takes into account backwards conflict, when

there are alternate sources for arrows.

A Trilogy program implementing this algorithm has been written. It is available and usable by anyone possessing a PC and the modestly priced Trilogy software. It is capable of verifying modest sized concurrent systems. Trilogy 2 will remove the ridiculous 64k limitation and have better garbage collection. Unfortunately, since Trilogy is not available for other, larger systems, this program must be viewed as a prototype.

This program has been tested with simple examples, networks of DME elements, and buffers of various sizes. Results of using behavior-state vs. execution-state encodings were compared, with the execution-state termination condition giving shorter execution time. We obtained empirical estimates of time and space complexity, in section 9.2, for two traditional asynchronous-circuit benchmarks, namely chains of buffer elements -- which are determinate systems, and rings of DME elements -- which are non-determinate systems. The buffer series verification cost is $O(n)$ in time and space. For the DME series, cost is $O(n^2)$ in space, and also $O(n^2)$ in time with an ideal implementation. In both cases, n is the number of components in the implementation.

The complexity of the algorithm is dependent on the structure of the system (i.e., mirror-coupled implementation) being verified. Complexity depends on the amount of

branching and recurrence in the closed system, on the number of nodes in the system stick figure, and on the shape of the search trees during non-trivial arrow checking. Visible choice is important for keeping the number of nodes small.

10.2 Future work

There is future work to be done and open problems to explore. Improvements can be made to the data structures encoding behavior automata, the algorithm and program. Then there is reprogramming in a widely used language, such as C. An open problem is for what wider classes of processes is this type of algorithm useful.

Sets of atom rules streamline the data structures by removing special socket matching machinery at some actions, but would require supplementary encoding should an action be a precondition of one action in one future, but of two actions in another future. Such a data structure is not presently included in the program.

In systems, process ports are physically linked together. The actions in neighboring processes, however, do not automatically correspond. During arrow checking, which is done by backing up along arrows in various processes, it possible, if one is not careful, to lose ones place in the system. At present the program assumes that linked actions are encoded with the same 'primes' (for example, + and -

transitions in a 4-cycle protocol). But it is reasonable, for example, to link a 4-cycle interface to a 2-cycle interface (such as a wire). At present, during arrow checking, if we were to pass to the 2-cycle process, we would lose our place in the 4-cycle process. We are careful not to do this, by using a 4-cycle encoding of the wire, for example.

The way out of this restriction might be a preprocessor to expand individual processes to match their neighbors' interfaces, or it might be to actually construct a system behavior automaton with its own data structure. This construction is possible while moving through the system actions. It would require that all the arrow checking be deferred until the system automaton was complete.

The program will have to be rewritten for efficiency and wider use. At the present time this means it should be written in a widely-used language such as C or LISP, in order to be portable to many different machines. The places where Trilogy uses **all** or **one** formulas to enter predicate (logical) mode, generally to allow backtracking, will have to be coded deterministicly. This will remove any unnecessary sorting. The only place where something like sorting is possibly required is in proc Arrowcheck when comparing presets. This should be done as efficiently as possible.

Other classes of processes include quasi-delay-

insensitive and delay-constrained.

The algorithm should work for quasi-delay-insensitive processes with the addition of isochronic fork specifications, allowing the algorithm to detect outputs from other processes (often gates) to handshake in place of silent transitions. An example of this is given by Probst and Li [4]. The main complication is establishing an "alias" for each silent transition. It should also be noted that one cannot just give a circuit diagram, showing AND gates, OR gates, etc. One must specify the uses to which each gate is put.

In delay-constrained systems, instead of a liveness requirement of "eventually", an enabled output is required within a specified time window, [min,max]. In a causal chain of arrows, the minimum and maximum times can be accumulated, just as the "active counts" are now accumulated during arrow checking, and compared with the requirements. Handshaking could be replaced by window timing assumptions. The general partial-order framework is a good starting point.

REFERENCES

[1] M. Nielsen, G. Plotkin and Glynn Winskel, Petri nets, event structures and domains, part I, Theoretical Computer Science 13 (1981), pp. 85-108.

[2] V. R. Pratt, Modeling concurrency with partial orders, Int. J. of Parallel Programming 15 (1986), pp. 33-71.

[3] A. Mazurkiewicz et al, Concurrent systems and inevitability, Theoretical Computer Science 64 (1989), pp. 281-304.

[4] D. K. Probst and H. F. Li, Abstract specification, composition and proof of correctness of delay-insensitive circuits and systems, Technical Report, Dept. of Computer Science, Concordia University, CS-VLSI-88-2. April 1988 (Revised March 1989).

[5] I. Sutherland, Micropipelines, CACM 32 (1989) pp. 720-738.

[6] A. J. Martin, The design of a self-timed circuit for distributed mutual exclusion, 1985 Chapel Hill conference on VLSI (1985) pp. 245-260.

[7] A. J. Martin, A synthesis method for self-timed VLSI circuits, Proc. 1987 ICCD, October 1987, pp. 224-229.

[8] D. L. Dill, Trace theory for automatic hierarchical verification of speed independent circuits, Proc. 1988 MIT conf. on VLSI, pp. 51-68.

[9] J. T. Udding, A formal model for defining and classifying delay-insensitive circuits and systems, Distributed Computing 1 (1986) pp. 197-204.

[10] J. Andrews, Trilogy, Complete Logic Systems, 1987

[11] P. Grogono, Review (of Trilogy), Computer Language, April 1988.

[12] Yu. V. Linnik, The Method of Least Squares, Pergamon Press, Oxford, pp. 174-176

[13] C.R.C. Standard Mathematical Tables, 11th edition, Chemical Rubber Publishing Company, Cleveland, Ohio, 1957.

[14] J. Staunstrup and M. R. Greenstreet, Designing delay insensitive circuits using "Synchronized transitions," Proc. International conference on applied formal methods for VLSI design, vol. 2, ed. Luc Claesen, pp 741-758, 1989.

[15] D. K. Probst and H. F. Li, Using partial-order semantics to avoid the state explosion problem in asynchronous systems, in E. M. Clarke and R. P. Kurshan, (Eds.) Workshop on Computer-Aided Verification '90, DIMACS Series, Vol. 3, 1991, pp. 15-24. Also Lecture Notes in Computer Science, Springer Verlag, forthcoming.

[16] D. K. Probst and L. C. Jensen, Controlling state explosion during automatic verification of delay-insensitive and delay-constrained VLSI systems using the POM verifier, 3rd NASA Symposium on VLSI Design, 1991.

APPENDIX 1.

POM USER'S MANUAL

Lin Jensen

POM: An automatic network verifier

USER'S GUIDE

Lin Jensen
August, 1991

CONTENTS

1. Requirements

POM is a program for automatic verification of delay-insensitive systems. To use it you need an IBM PC or compatible. A hard disk is recommended. You also need Trilogy, version 1.3, available from

Complete Logic Systems
741 Blueridge Ave.
North Vancouver, B.C.
Canada, V7R 2J5

and the disk of POM modules.

2.a. Installing Trilogy

Install Trilogy as recommended in the manual, creating a directory named trilogy on your hard disk, moving to that directory, and running the program **install** found on trilogy diskette I.

2.b. Installing POM

Copy all the files on the POM distribution diskette to the trilogy directory, for example:

```
cd \trilogy
copy a:*.*
```

Run Trilogy:

```
trilogy
```

Wait until it says Trilogy ready, then select OPTIONS at the top. From the Options menu which appears, select Restore library from file. The file name is LIBRARY.POM. This library catalogues the Trilogy system modules, the examples that come with the Trilogy distribution, and the POM modules. (If you are already using Trilogy, be sure to save your own library to a file first.) Escape (Esc key) back to the command bar at the top of the screen. From here you can exit to DOS, or open MODULES.

You can now henceforth issue queries to POM, since it is catalogued in the Trilogy library.

2.c. If you have a 2 floppy disk system

First, make a backup copy of the POM diskette. Then install Trilogy as recommended in the manual. When you run Trilogy, after you start it, you can take out the diskette containing TRILOGY.EXE and replace it with the POM diskette.

Caveat: I have never done this.

2.d. A sample run.

From the command bar at the top of the Trilogy screen (see Trilogy manual, section 1.5 Running Trilogy, page xvii) select MODULES. A window opens in the middle of the screen, saying "Enter module name". Type

pom

and press Enter. (Note all lower case.)

Wait while all the modules are opened.

Move to the QUERY window near the top of the screen by pressing left arrow. Type in the query:

```
SetBuf (2)
```

and press F2. This executes the query, which automatically sets up a network of 2 single-transition buffers, to check against the specification of a 2-transition buffer (2-Buffer).

After this query is finished, get back to the query window by pressing Q or q or Enter. Now type in the query:

```
NetVerify ('Sample run')
```

nd press F2. This query always verifies the most recently set up system, using the title you give as parameter.

3. Using the POM queries

Run Trilogy:

```
cd \trilogy
trilogy
```

Wait until it says Trilogy ready, then select MODULES. A window opens in the middle of the screen, saying "Enter module name". Type

pom

and press Enter. Note all lower case. Case is significant.

Wait while all the modules are opened. The module pom actually contains no code, its purpose is to open all the other modules. Its source consists of a general comment outlining the queries you would possibly use, which are contained in the modules enterp, setup, and netverify.

To execute a query, you need to get into the query window near the top of the screen, via the command bar QUERY option. (Or directly from MODULES as mentioned previously. In Trilogy, the bottom of the screen always indicates what you can currently do.) You type in your query, and press F2 to execute it. After conclusion, you are back at the command bar.

There are three sorts of queries available in POM.

Module enterp provides services for entering new process definitions, revising them, and maintaining the process database, ProcessF.

The query:

EnterProcess ('My favorite process')

allows you to enter the process definition for a process named 'My favorite process'. You will be asked to fill in the data structure for a process.

A word about syntax. Port names and arrow labels are strings. Strings are delimited by single quotes. Two single quotes in a row is the empty string. However, if your string meets the requirements for a variable name, the quotes can be omitted. Text being entered can be edited, when finished, press F2 to accept it (not Enter).

First you specify each 'atomic' grammar rule. Each one consists of a port name, then select 0 or more primes, plain, loop or choice 'marking', and enter the lists of needs, gives, and tickles. (Tickles is normally an empty list, except for the DME element.) For example, if a rule needs the arrows alpha and beta, you type: alpha,beta, and press F2. The screen shows: alpha,beta,Nil (the Nil, indicating end of list, was already there). For an empty tickles list, just press F2. The screen shows: Nil . When there are no more rules, enter two single quotes as port name (the empty string).

Now all the port names will be presented to you, one at a time, and you select their type, In or Out.

Finally, all the arrow names are presented to you in a list, with counts of 0. You edit this list to reflect the desired initial state. For example, suppose the initial state consists of one alpha arrow and two free arrows. The list looks like:

(alpha,0),(beta,0),(free,0),(message,0),Nil

edit it to read:

(alpha,01),(beta,0),(free,02),(message,0),Nil

and press F2. Entering this process is now complete.

The query:

ModState (old:<8, new:<8)

where the old name exists and the new name does not already exist, allows you to create a new process which differs from the old one only in its initial state. The list of arrows and their counts in the initial state of old appears, you edit the counts of arrows using normal editing keys, then press F2.

For example, a 6-transition buffer specification can be made from a (one-transition) buffer specification using the query:

ModState ('Buffer', '6-Buffer')

which shows a list of arrows including ...,(free,1),...,Nil.

Edit this to read ...(free,6),...,Nil and press F2. The 6-Buffer specification is now in ProcessF.

The query:

ReviseP (1)

goes through the process database file ProcessF and lets you revise it with Keep, Change and Delete. Change lets you modify any part of a process definition.

The query:

ArbiterSpec (n)

creates a specification for an n-Arbiter, n in the range 2-10. (It will fail if the specification already exists.)

Module **setup** has queries to set up a network to verify, which is then stored in the (relatively temporary) database file NetF. There are also special automated setups for DME rings of DME elements implementing arbiters, and single buffers implementing a multiple-transition buffer.

During these queries, some time is taken converting process representations from external form to internal form. Essentially, strings are being converted to predetermined scalar types, and lists to fixed length arrays. This is for efficiency, and so that sizes can be changed without affecting the external representation on ProcessF.

The query:

Setup (1)

sets up any network interactively. You are asked to select each process from the list of known processes. First the specification (mirror), then the implementation processes. Currently 10 implementation processes are allowed, select the process NULL to fill unused slots.

Next you specify the links between processes. For each output port, you select a process and input port (that is not already connected). Sometimes there is only one port left, then it is selected by default.

The query:

SetupDME (n)

sets up a ring of n DME elements (n = 2..10) without any user interaction. Similarly, the query:

SetBuf (n)

sets up a system of n single-transition buffers.

Module **netverify** contains the query:

NetVerify ('Title of this verification') & Dates (date)

which verifies the last network set up (which is saved in the file NetF), and includes today's date (which is of course optional).

Output can be redirected from the screen to a file, for example the query:

```
in 'a:myoutput.txt' Netverify ('My title')
```

carries out the latest verification as above, with output redirected to file MYOUTPUT.TXT on the floppy disk in drive A:.

4. Recompiling modules.

Modules consist of a source file (.L extension) and a compiled file (.MOD extension). In the module list, s indicates the source file is open and available for editing, o indicates that the compiled object code is available. You should know that Trilogy is sensitive to the date and time of compilation of modules. Whenever a compilation becomes obsolete, the o is replaced by a *. The module must be recompiled by highlighting its name and pressing F2. Used modules must be compiled before the modules that use them. Error messages usually point out the required (partial!) order.

To open a source file, highlight the module name, and press Enter. The source is now in the edit window, and you can view and edit it using standard editor keys. (If you make any inadvertent changes, choose "Ignore changes" when closing the source or leaving Trilogy.) To return to the MODULES window, press Esc. There is a limit to editor memory space, so not to many source files can be open at once.

One module that you may wish to modify is **netverify_t**, in order to reconfigure the number of implementation processes, number of ports per process, or number of arrows per process. In each case, both a constant and a type must be redefined, and kept compatible. Unfortunately, this requires recompiling most of the modules.

An additional complication arises with the module **packing** since its code is written in assembly language. After compiling the module packing, you must exit Trilogy, and run the program linkmod. In response to the prompts, packing is the name of the .mod file, and packing is also the name of the .obj file. The path to Trilogy is the current directory, so just press Enter. Now restart Trilogy and continue compiling.

APPENDIX 2.

SAMPLE POM PROGRAM OUTPUT

a. Correct systems

b. Bugs found in proposed implementations

Lin Jensen

```
Network verfication for:
     4-Buffer implemented by 4 single-transition buffers

Processes       ----- Ports -----
Pm   4-Buffer     a,b,c,d,
P1   Buffer     a,b,c,d,
P2   Buffer     a,b,c,d,
P3   Buffer     a,b,c,d,
P4   Buffer     a,b,c,d,     -----> Hash # 1  <---
-++++-> Starting spec command   a  c  d  b  <-++++-
          event pair  Pm, a =  P1, a
     ... moving
          event pair  P1, c =  P2, a
          event pair  P1, d =  Pm, d
Found event a
Found event a
Found event a
Found event a
Found event b
noncausal = {(a,0),(b,3),},Nil
causal =    {(a,0),(b,3),},Nil
     ... moving
          event pair  P2, c =  P3, a
          event pair  P2, d =  P1, b
     ... moving
          event pair  P3, c =  P4, a
          event pair  P3, d =  P2, b
     ... moving
          event pair  P4, c =  Pm, c
Found event b
Found event b
Found event b
Found event b
Found event a
noncausal = {(a,0),(b,0),},Nil
causal =    {(a,0),(b,0),},Nil
          event pair  P4, d =  P3, b
     ... moving
          event pair  Pm, b =  P4, b
==== found current state in hashlist====== entry # 1
Verification suceeded!
date = 07/04/1991
Success
elapsed time: 00:00:14.34   start: 14:23:46.81   end: 14:24:01.15
```

```
Network verfication for: (passive) buffer inplemented by
                         C-element & Fork

Processes       ----- Ports -----
Pm    Passive buffer    a,b,c,d,
P1    C-element    a,b,c,
P2    Fork    a,b,c,    -----> Hash # 1  <---
-++++-> Starting spec command   a  b  c  d  <-++++-
         event pair  Pm, b =  P1, b
Found event c
noncausal = {(c,0),},Nil
causal =    {(c,0),},Nil
         event pair  Pm, a =  P1, a
Found event c
noncausal = {(c,0),},Nil
causal =    {(c,0),},Nil
    ... moving
         event pair  P1, c =  P2, a
Found event b
Found event c
Found event c
Found event b
noncausal = {(b,0),(c,0),},Nil
causal =    {(b,0),(c,0),},Nil
    ... moving
         event pair  P2, b =  Pm, d
Found event a
Found event b
noncausal = {(a,0),(b,0),},Nil
causal =    {(a,0),(b,0),},Nil
         event pair  P2, c =  Pm, c
Found event a
Found event b
noncausal = {(a,0),(b,0),},Nil
causal =    {(a,0),(b,0),},Nil
==== found current state in hashlist====== entry # 1
Verification suceeded!
date = 07/04/1991
Success
elapsed time: 00:00:12.30   start: 15:28:13.73   end: 15:28:26.03
```

```
Network verfication for: 2-Arbiter by ring of 2 DME elements

Processes       ----- Ports -----
Pm   2-Arbiter    b,q,a,p,
P1   LDME    la,lr,ra,rr,ua,ur,
P2   DME    la,lr,ra,rr,ua,ur,
    ... moving
         event pair  Pm, b =  P2, ur
         event pair  Pm, a =  P1, ur
    ... moving
         event pair  P2, rr =  P1, lr    -----> Hash # 1  <---
-------- START FORK ----
   ...deferring...  P1, la =  P2, ra
   ...deferring...  P1, ua =  Pm, p
-++++-> Starting spec command   p  a'  p'  a  <-++++-
         event pair  P1, ua =  Pm, p
Found event a'
Found event a'
Found event b'
Found event a'
Found event a
noncausal = {(b',0),(a,0),},{(a,0),(a',0),},Nil
causal =    {(b',0),(a,0),},{(a,0),(a',0),},Nil
    ... moving
         event pair  Pm, a' =  P1, ur'
    ... moving
         event pair  P1, ua' =  Pm, p'
    ... moving
         event pair  Pm, a =  P1, ur
==== found current state in hashlist====== entry # 1
         event pair  P1, la =  P2, ra
    ... moving
         event pair  P1, rr =  P2, lr    -----> Hash # 2  <---
-------- START FORK ----
   ...deferring...  P2, la =  P1, ra
   ...deferring...  P2, ua =  Pm, q
-++++-> Starting spec command   q  b'  q'  b  <-++++-
         event pair  P2, ua =  Pm, q
Found event b'
Found event b'
Found event a'
Found event b'
Found event b
noncausal = {(b,0),(b',0),},{(b,0),(a',0),},Nil
causal =    {(b,0),(b',0),},{(b,0),(a',0),},Nil
    ... moving
         event pair  Pm, b' =  P2, ur'
    ... moving
         event pair  P2, ua' =  Pm, q'
    ... moving
         event pair  Pm, b =  P2, ur
==== found current state in hashlist====== entry # 2
```

```
          event pair  P2, la =  P1, ra
     ... moving
          event pair  P2, rr =  P1, lr
==== found current state in hashlist====== entry # 1
Verification suceeded!
date = 05/18/1991
Success
elapsed time: 00:00:18.57   start: 14:11:35.86   end: 14:11:54.43



Network verfication for: 3-Arbiter implemented by 3 DME elements

Processes       ----- Ports -----
Pm   3-Arbiter    c,r,b,q,a,p,
P1   LDME    la,lr,ra,rr,ua,ur,
P2   DME    la,lr,ra,rr,ua,ur,
P3   DME    la,lr,ra,rr,ua,ur,
     ... moving
          event pair  Pm, c =  P3, ur
          event pair  Pm, b =  P2, ur
          event pair  Pm, a =  P1, ur
     ... moving
          event pair  P2, rr =  P1, lr
          event pair  P3, rr =  P2, lr    -----> Hash # 1  <---
-------- START FORK ----
   ...deferring...  P1, la =  P2, ra
   ...deferring...  P1, ua =  Pm, p
-++++-> Starting spec command   p  a'  p'  a  <-++++-
          event pair  P1, ua =  Pm, p
Found event a'
Found event b'
Found event a'
Found event c'
Found event b'
Found event a'
Found event a
noncausal = {(c',0),(a,0),},{(b',0),(a,0),},{(a,0),(a',0),},Nil
causal =    {(c',0),(a,0),},{(b',0),(a,0),},{(a,0),(a',0),},Nil
     ... moving
          event pair  Pm, a' =  P1, ur'
     ... moving
          event pair  P1, ua' =  Pm, p'
     ... moving
          event pair  Pm, a =  P1, ur
==== found current state in hashlist====== entry # 1
          event pair  P1, la =  P2, ra
     ... moving
          event pair  P1, rr =  P3, lr    -----> Hash # 2  <---
-------- START FORK ----
   ...deferring...  P2, la =  P3, ra
```

```
   ...deferring...  P2, ua =  Pm, q
-++++-> Starting spec command   q  b'  q'  b  <-++++-
        event pair  P2, ua =  Pm, q
Found event b'
Found event c'
Found event b'
Found event a'
Found event c'
Found event b'
Found event b
noncausal = {(c',0),(b,0),},{(b,0),(b',0),},{(b,0),(a',0),},Nil
causal =    {(c',0),(b,0),},{(b,0),(b',0),},{(b,0),(a',0),},Nil
    ... moving
        event pair  Pm, b' =  P2, ur'
    ... moving
        event pair  P2, ua' =  Pm, q'
    ... moving
        event pair  Pm, b =  P2, ur
==== found current state in hashlist====== entry # 2
        event pair  P2, la =  P3, ra
    ... moving
        event pair  P2, rr =  P1, lr    -----> Hash # 3  <---
-------- START FORK ----
   ...deferring...  P3, la =  P1, ra
   ...deferring...  P3, ua =  Pm, r
-++++-> Starting spec command   r  c'  r'  c  <-++++-
        event pair  P3, ua =  Pm, r
Found event c'
Found event a'
Found event c'
Found event b'
Found event a'
Found event c'
Found event c
noncausal = {(c,0),(c',0),},{(c,0),(b',0),},{(c,0),(a',0),},Nil
causal =    {(c,0),(c',0),},{(c,0),(b',0),},{(c,0),(a',0),},Nil
    ... moving
        event pair  Pm, c' =  P3, ur'
    ... moving
        event pair  P3, ua' =  Pm, r'
    ... moving
        event pair  Pm, c =  P3, ur
==== found current state in hashlist====== entry # 3
        event pair  P3, la =  P1, ra
    ... moving
        event pair  P3, rr =  P2, lr
==== found current state in hashlist====== entry # 1
Verification suceeded!
Success
elapsed time: 00:00:28.79   start: 13:10:20.30   end: 13:10:49.09
```

```
Network verfication for: 2-4-arbiter by ring of DME elements

Processes       ----- Ports -----
Pm   2-4-Arbiter    d,s,c,r,b,q,a,p,
P1   LDME    la,lr,ra,rr,ua,ur,
P2   DME    la,lr,ra,rr,ua,ur,
P3   LDME    la,lr,ra,rr,ua,ur,
P4   DME    la,lr,ra,rr,ua,ur,
    ... moving
         event pair  Pm, d =  P4, ur
         event pair  Pm, c =  P3, ur
         event pair  Pm, b =  P2, ur
         event pair  Pm, a =  P1, ur
    ... moving
         event pair  P2, rr =  P1, lr
         event pair  P4, rr =  P3, lr    -----> Hash # 1  <---
-------- START FORK ----
   ...deferring...  P1, la =  P2, ra
   ...deferring...  P1, ua =  Pm, p
   ...deferring...  P3, la =  P4, ra
   ...deferring...  P3, ua =  Pm, r
-++++-> Starting spec command   r  c'  r'  c  <-++++-
         event pair  P3, ua =  Pm, r
Found event c'
Found event a'
Found event d'
Found event c'
Found event b'
Found event a'
Found event d'
Found event c'
Found event c
noncausal = {(d',1),(c,0),},{(c,0),(c',1),},{(c,0),(b',1),},
{(c,0),(a',1),},Nil
causal =    {(d',1),(c,0),},{(c,0),(c',0),},{(c,0),(c',1),},
{(c,0),(b',0),},{(c,0),(a',1),},Nil
    ... moving
         event pair  Pm, c' =  P3, ur'
    ... moving
         event pair  P3, ua' =  Pm, r'
    ... moving
         event pair  Pm, c =  P3, ur
==== found current state in hashlist====== entry # 1
         event pair  P3, la =  P4, ra
    ... moving
         event pair  P3, rr =  P2, lr    -----> Hash # 2  <---
-------- START FORK ----
   ...deferring...  P1, la =  P2, ra
   ...deferring...  P1, ua =  Pm, p
   ...deferring...  P4, ua =  Pm, s
-++++-> Starting spec command   s  d'  s'  d  <-++++-
         event pair  P4, ua =  Pm, s
```

```
Found event d'
Found event b'
Found event a'
Found event d'
Found event c'
Found event b'
Found event a'
Found event d'
Found event d
noncausal = {(d,0),(d',1),},{(d,0),(c',1),},{(d,0),(b',1),},
{(d,0),(a',1),},Nil
causal =    {(d,0),(d',0),},{(d,0),(d',1),},{(d,0),(c',0),},
{(d,0),(b',0),},{(d,0),(a',1),},Nil
    ... moving
        event pair  Pm, d' =  P4, ur'
    ... moving
        event pair  P4, ua' =  Pm, s'
    ... moving
        event pair  Pm, d =  P4, ur
==== found current state in hashlist====== entry # 2
-++++-> Starting spec command   p  a'  p'  a  <-++++-
        event pair  P1, ua =  Pm, p
Found event a'
Found event c'
Found event b'
Found event a'
Found event d'
Found event c'
Found event b'
Found event a'
Found event a
noncausal = {(d',1),(a,0),},{(c',1),(a,0),},{(b',1),(a,0),},
{(a,0),(a',1),},Nil
causal =    {(d',1),(a,0),},{(c',1),(a,0),},{(b',1),(a,0),},
{(a,0),(a',0),},{(a,0),(a',1),},Nil
    ... moving
        event pair  Pm, a' =  P1, ur'
    ... moving
        event pair  P1, ua' =  Pm, p'
    ... moving
        event pair  Pm, a =  P1, ur
==== found current state in hashlist====== entry # 2
        event pair  P1, la =  P2, ra
    ... moving
        event pair  P1, rr =  P4, lr    -----> Hash # 3  <---
-------- START FORK ----
   ...deferring...  P2, la =  P3, ra
   ...deferring...  P2, ua =  Pm, q
   ...deferring...  P4, la =  P1, ra
   ...deferring...  P4, ua =  Pm, s
-++++-> Starting spec command   s  d'  s'  d  <-++++-
        event pair  P4, ua =  Pm, s
```

```
Found event d'
Found event b'
Found event a'
Found event d'
Found event c'
Found event b'
Found event a'
Found event d'
Found event d
noncausal = {(d,0),(d',1),},{(d,0),(c',1),},{(d,0),(b',1),},
{(d,0),(a',1),},Nil
causal =    {(d,0),(d',0),},{(d,0),(d',1),},{(d,0),(c',0),},
{(d,0),(b',1),},{(d,0),(a',1),},Nil
    ... moving
        event pair  Pm, d' =  P4, ur'
    ... moving
        event pair  P4, ua' =  Pm, s'
    ... moving
        event pair  Pm, d =  P4, ur
==== found current state in hashlist====== entry # 3
        event pair  P4, la =  P1, ra
    ... moving
        event pair  P4, rr =  P3, lr    -----> Hash # 4  <----
-------- START FORK ----
   ...deferring...  P1, ua =  Pm, p
   ...deferring...  P2, la =  P3, ra
   ...deferring...  P2, ua =  Pm, q
-++++-> Starting spec command   q  b'  q'  b  <-++++-
        event pair  P2, ua =  Pm, q
Found event b'
Found event d'
Found event c'
Found event b'
Found event a'
Found event d'
Found event c'
Found event b'
Found event b
noncausal = {(d',1),(b,0),},{(c',1),(b,0),},{(b,0),(b',1),},
{(b,0),(a',1),},Nil
causal =    {(d',1),(b,0),},{(c',1),(b,0),},{(b,0),(b',0),},
{(b,0),(b',1),},{(b,0),(a',1),},Nil
    ... moving
        event pair  Pm, b' =  P2, ur'
    ... moving
        event pair  P2, ua' =  Pm, q'
    ... moving
        event pair  Pm, b =  P2, ur
==== found current state in hashlist====== entry # 4
        event pair  P2, la =  P3, ra
    ... moving
        event pair  P2, rr =  P1, lr
```

```
==== found current state in hashlist====== entry # 1
-++++-> Starting spec command   p  a'  p'  a  <-++++-
         event pair  P1, ua =  Pm, p
Found event a'
Found event c'
Found event b'
Found event a'
Found event d'
Found event c'
Found event b'
Found event a'
Found event a
noncausal = {(d',1),(a,0),},{(c',1),(a,0),},{(b',1),(a,0),},
{(a,0),(a',1),},Nil
causal =    {(d',0),(a,0),},{(c',0),(a,0),},{(b',1),(a,0),},
{(a,0),(a',0),},{(a,0),(a',1),},Nil
    ... moving
         event pair  Pm, a' =  P1, ur'
    ... moving
         event pair  P1, ua' =  Pm, p'
    ... moving
         event pair  Pm, a =  P1, ur
==== found current state in hashlist====== entry # 4
-++++-> Starting spec command   q  b'  q'  b  <-++++-
         event pair  P2, ua =  Pm, q
Found event b'
Found event d'
Found event c'
Found event b'
Found event a'
Found event d'
Found event c'
Found event b'
Found event b
noncausal = {(d',1),(b,0),},{(c',1),(b,0),},{(b,0),(b',1),},
{(b,0),(a',1),},Nil
causal =    {(d',1),(b,0),},{(c',1),(b,0),},{(b,0),(b',0),},
{(b,0),(b',1),},{(b,0),(a',0),},Nil
    ... moving
         event pair  Pm, b' =  P2, ur'
    ... moving
         event pair  P2, ua' =  Pm, q'
    ... moving
         event pair  Pm, b =  P2, ur
==== found current state in hashlist====== entry # 3
         event pair  P2, la =  P3, ra
    ... moving
         event pair  P2, rr =  P1, lr    -----> Hash # 5  <---
-------- START FORK ----
   ...deferring...  P3, ua =  Pm, r
   ...deferring...  P4, la =  P1, ra
   ...deferring...  P4, ua =  Pm, s
```

```
-++++-> Starting spec command   s  d'  s'  d  <-++++-
         event pair  P4, ua =  Pm, s
Found event d'
Found event b'
Found event a'
Found event d'
Found event c'
Found event b'
Found event a'
Found event d'
Found event d
noncausal = {(d,0),(d',1),},{(d,0),(c',1),},{(d,0),(b',1),},
{(d,0),(a',1),},Nil
causal =    {(d,0),(d',0),},{(d,0),(d',1),},{(d,0),(c',1),},
{(d,0),(b',1),},{(d,0),(a',1),},Nil
    ... moving
         event pair  Pm, d' =  P4, ur'
    ... moving
         event pair  P4, ua' =  Pm, s'
    ... moving
         event pair  Pm, d =  P4, ur
==== found current state in hashlist====== entry # 5
         event pair  P4, la =  P1, ra
    ... moving
         event pair  P4, rr =  P3, lr
==== found current state in hashlist====== entry # 1
-++++-> Starting spec command   r  c'  r'  c  <-++++-
         event pair  P3, ua =  Pm, r
Found event c'
Found event a'
Found event d'
Found event c'
Found event b'
Found event a'
Found event d'
Found event c'
Found event c
noncausal = {(d',1),(c,0),},{(c,0),(c',1),},{(c,0),(b',1),},
{(c,0),(a',1),},Nil
causal =    {(d',1),(c,0),},{(c,0),(c',0),},{(c,0),(c',1),},
{(c,0),(b',0),},{(c,0),(a',0),},Nil
    ... moving
         event pair  Pm, c' =  P3, ur'
    ... moving
         event pair  P3, ua' =  Pm, r'
    ... moving
         event pair  Pm, c =  P3, ur
==== found current state in hashlist====== entry # 5
-++++-> Starting spec command   p  a'  p'  a  <-++++-
         event pair  P1, ua =  Pm, p
Found event a'
Found event c'
```

```
Found event b'
Found event a'
Found event d'
Found event c'
Found event b'
Found event a'
Found event a
noncausal = {(d',1),(a,0),},{(c',1),(a,0),},{(b',1),(a,0),},
{(a,0),(a',1),},Nil
causal =    {(d',0),(a,0),},{(c',1),(a,0),},{(b',1),(a,0),},
{(a,0),(a',0),},{(a,0),(a',1),},Nil
    ... moving
        event pair  Pm, a' =  P1, ur'
    ... moving
        event pair  P1, ua' =  Pm, p'
    ... moving
        event pair  Pm, a =  P1, ur
==== found current state in hashlist====== entry # 1
        event pair  P1, la =  P2, ra
    ... moving
        event pair  P1, rr =  P4, lr    ------> Hash # 6  <---
-------- START FORK ----
   ...deferring...  P2, ua =  Pm, q
   ...deferring...  P3, la =  P4, ra
   ...deferring...  P3, ua =  Pm, r
-++++-> Starting spec command   r  c'  r'  c  <-++++-
        event pair  P3, ua =  Pm, r
Found event c'
Found event a'
Found event d'
Found event c'
Found event b'
Found event a'
Found event d'
Found event c'
Found event c
noncausal = {(d',1),(c,0),},{(c,0),(c',1),},{(c,0),(b',1),},
{(c,0),(a',1),},Nil
causal =    {(d',1),(c,0),},{(c,0),(c',0),},{(c,0),(c',1),},
{(c,0),(b',1),},{(c,0),(a',1),},Nil
    ... moving
        event pair  Pm, c' =  P3, ur'
    ... moving
        event pair  P3, ua' =  Pm, r'
    ... moving
        event pair  Pm, c =  P3, ur
==== found current state in hashlist====== entry # 6
        event pair  P3, la =  P4, ra
    ... moving
        event pair  P3, rr =  P2, lr
==== found current state in hashlist====== entry # 3
-++++-> Starting spec command   q  b'  q'  b  <-++++-
```

```
         event pair  P2, ua =  Pm, q
Found event b'
Found event d'
Found event c'
Found event b'
Found event a'
Found event d'
Found event c'
Found event b'
Found event b
noncausal = {(d',1),(b,0),},{(c',1),(b,0),},{(b,0),(b',1),},
{(b,0),(a',1),},Nil
causal =    {(d',0),(b,0),},{(c',1),(b,0),},{(b,0),(b',0),},
{(b,0),(b',1),},{(b,0),(a',0),},Nil
    ... moving
         event pair  Pm, b' =  P2, ur'
    ... moving
         event pair  P2, ua' =  Pm, q'
    ... moving
         event pair  Pm, b =  P2, ur
==== found current state in hashlist====== entry # 6
Verification suceeded!
05/18/1991
date = 05/18/1991
Success
elapsed time: 00:01:46.83   start: 14:03:32.02    end: 14:05:18.85
```

b. Runs where bugs were found in systems

```
Network verfication for: (passive) Buffer implemented by 2 Wires

Processes       ----- Ports -----
Pm   Passive buffer    a,b,c,d,
P1   2-wires    a,b,c,d,    -----> Hash # 1  <---
-++++-> Starting spec command   a  b  c  d  <-++++-
          event pair  Pm, b =  P1, c
          event pair  Pm, a =  P1, a
    ... moving
          event pair  P1, b =  Pm, d
Found event a
noncausal = {(a,0),(b,0),},Nil
causal =    {(a,0),},Nil
SAFETY/LIVENESS VIOLATION (arrows) at Pm, d
Failure
elapsed time: 00:00:06.97   start: 15:39:51.18   end: 15:39:58.15



Network verfication for: 2 wires implemented by (passive) buffer

Processes       ----- Ports -----
Pm   2-wires    a,b,c,d,
P1   Passive buffer    a,b,c,d,    -----> Hash # 1  <---
-++++-> Starting spec command   a  c  b  d  <-++++-
          event pair  Pm, c =  P1, b
          event pair  Pm, a =  P1, a
    ... moving
          event pair  P1, c =  Pm, d
Found event c
Found event a
noncausal = {(c,0),},Nil
causal =    {(a,0),(c,0),},Nil
SAFETY/LIVENESS VIOLATION (arrows) at Pm, d
Failure
elapsed time: 00:00:06.92   start: 15:18:40.42   end: 15:18:47.34
```

```
Network verfication for: 2 wires implemented by an arbiter

Processes       ----- Ports -----
Pm   2-Wires(4 cycle)    a,b,p,q,
P1   2-Arbiter    b,q,a,p,     -----> Hash # 1  <---
-+++-> Starting spec command   a  b  p  q  a'  b'  p'  q'<-+++
         event pair  Pm, b =  P1, b
         event pair  Pm, a =  P1, a     -----> Hash # 2  <---
ERROR OF CHOICE IN THE MIDDLE OF SPEC COMMAND
Failure
elapsed time: 00:00:07.14   start: 19:05:03.77   end: 19:05:10.91


Network verfication for: Arbiter implemented by 2 wires

Processes       ----- Ports -----
Pm   2-Arbiter    b,q,a,p,
P1   2-Wires(4 cycle)    a,b,p,q,
    ... moving
         event pair  Pm, b =  P1, b
         event pair  Pm, a =  P1, a     -----> Hash # 1  <---
+++-> ERROR, unable to find command containing these
      spec. actions:
        q  p
 --- among these commands:
       p  a'  p'  a
       q  b'  q'  b
         event pair  P1, q =  Pm, q
Found event b
noncausal = {(b,0),(b',0),},{(b,0),(a',0),},Nil
causal =    {(b,0),},Nil
SAFETY/LIVENESS VIOLATION (arrows) at Pm, q
Failure
elapsed time: 00:00:06.92   start: 10:01:22.97   end: 10:01:29.89
```

APPENDIX 3.

POM PROGRAM LISTING

September 1991

Lin Jensen

(please note: PRODUCTION is the old term for COMMAND

POM: The program
Delay-insensitive verification
Lin Jensen

this is module process_t type declarations
used by enterp ProcessF setup
also by: netverify_t
(for types Mark, Prime, Role, NumberArrows)

January 29, 1990
November 1990 : Arrow "Tickling"

Declarations exported to main verification :

)
Mark = Plain | Loop | Choice
(starts of Generators are marked
to indicate looping or choice cuts
-- used for hashing and forking of alg.)
NumberArrows = [0..255]
(state may have 0,1 or more arrows of same type)
Role = In | Out | Unused
Prime = Noprime | Oneprime | Prime2 | Prime3 (needs 2 bits)

(====== EXTERNAL VIEW OF PROCESS(ES) ====== use strings ========
this is independent of the arbitrary fixed size of arrow
vectors and portnames used in the actual verfication procedures
==========)

ExtRole = list (S(port), Role) (designates each port as In/Out)

InitialState = list (S(arrow), NumberArrows)
(multiset of the arrows "from initialization" -> initial events)

ExtAct = (S(portname), [0..](Prime)
(a semantic action class is not simply all the actions at a
given port. Events at a port may play a different role in
the partial order, generally when their semantics are
different. We choose to distinguish these with a number of
primes, which will be type-cast into the type Prime)

ExtArrows = list S (a set of arrows as names)

ExtRule = (extact:ExtAct, mark:Mark, extneeds:ExtArrows,
extgives:ExtArrows, extticKles:ExtArrows)
(also needs one of the tickle arrows, without using it up.)

ExtRules = list ExtRule

```
(--will be converted into the internal form  Process, see
                                                    netverify_t--)

ExtProcess = (name:S, role:ExtRole, initial:InitialState,
                  extrules:ExtRules, extcoupled:list list ExtAct )

   {coupled choice. Each list contains choice events which must
    be taken together -- there is a choice of generators with
    non-sequential concatenation.  It only makes sense to have at
    least two actions in a list.  All must be marked as Choice
    and all must be in the same Role (In/Out)}

Processf = file ExtProcess     {Database of interesting processes}

(------------------- end of module process_t ------------------)
```

```
{-------------------------------------------------------------
module: enterp
   for: entering a process into database ProcessF in its
        external view
  uses: process_t files windows
-------------------------------------------------------------
        Lin Jensen
        January 29, 1990
        Delay insensitive verification
        November, 1990: Tickle arrows implemented
```

also to make new process which is a modification fo the initial state of an already entered process:

```
      subr ModState (old:<S, new:<S) iff
            old name exists, the new name does not already exist

      subr ReviseP (1)
               {revise the processfile with Keep, Change Delet-}

      proc ArbiterSpec (n)          {create spec for n-Arbiter}
-------------------------------------------------------------}

subr EnterProcess (name:<S) iff

{Enter a Process description from screen and store in database
file ProcessF in external form, that is, strings, independent of
number of port or arrow names.  Store it with this name, which
must not exist on the file}

   one x x = 1 & ProcessF(name,_,_,_,_) end &
                                        {not a duplicate name}
                                                {enter rules}
   EnterRules (rules) &
                                                {enter port roles}
   EnterRoles (rules, roles) &
                                        {enter initial state}
   EnterState (rules, initstate) &
                                {enter sets of coupled actions}
   {--------- input not yet implemented ------ not yet used in
                                                    netverify ---}

   process:>ExtProcess &
   process = (name, roles, initstate, rules, Nil{coupled}) &
   Insert (ProcessF, process)

{-------------------------------------------------------------}
                        {enter rules}
```

```
subr   EnterRules (rules:>ExtRules) iff
    {== instructions ==}
    Putw (1,Nl,'Now you will enter the rules for this process.',
           Nl,'Each rule starts with a port name (any string).',
           Nl,'   Enter '''' when done with rules.',
           Nl,
  ' Then pick "primes" and "Plain/Loop/Choice" in popup windows',
      Nl,'   List the arrows (strings) needed',
      Nl,'   List the arrows given.   (comma after each arrow)',
      Nl,'   List any alternate enabling arrows.' ) &
    EnterRule (Nil, rules)

subr EnterRule (sofar:<ExtRules, rules:>ExtRules) iff
   {port}
   InString ('Port name for this rule :',port) &
   if port = '' then   (no more rules, done)
      rules = sofar
   else
      {prime, mark}
      Popw (3, 20, Dpopattr, ['0 - no prime','1 - ''',
              '2 - ''''', '3 - ''''''','4 - ''''''''], prime) &
      Pprime (1,prime) &
      Popw (5, 30, Dpopattr, ['Plain (not start of command)',
                              'Loop (at start of command)',
                              'Choice event'], marki) &
      mark = marki:Mark &
      {needs}
      InSlist ('Needs: arrows required for this event', needs) &
      {gives}
      InSlist ('Gives: arrows starting from this event', gives) &
      {tickles}
      InSlist
         ('Tickle: alternate enabling arrows passing this event',
                tickles) &
      newrule = ((port,prime),mark,needs,gives,tickles):ExtRule &
      EnterRule ((newrule,sofar), rules)
   end

local proc Pprime (pid:<I, p:<[0..]) iff
                        {print the corresponding number of primes}
  if p <> 0 then
     Putw (pid,'''') &
     Pprime (pid, p-1)
  end

{-------------------------------------------------------------}
```

```
                          {enter port roles}
subr   EnterRoles (rules:<ExtRules, roles:>ExtRole) iff
                          {make list of port names &}
     portnames:>list S &
     all port in portnames
         rule in rules &
         rule.extact = (port,_)
     end &
     {======= can fancy this up by opening a window here=======}
     Putw (1,Nl,
'As each port name appears, chose In/Out in the popup window') &
                                          {each one is In or Out}
     EnterRole (portnames, Nil, roles)

subr  EnterRole (ports:<list S, sofar:< ExtRole, roles:>ExtRole)
                iff
   if ports = Nil then
      roles = sofar       {finished}
   else
      port = ports.h &
      Putw (1, Nl, '    ', port) &
      Popw (3, 20, Dpopattr, ['In','Out'], answer) &
      rol = answer:Role &
      Putw (1, '   ',rol) &
      newlist:>ExtRole &
      newlist = (port,rol),Nil &
      EnterRole (ports.t, Append(sofar,newlist), roles)
   end

{-------------------------------------------------------------}
                          {enter initial state}
subr   EnterState (rules:<ExtRules, initstate:>InitialState) iff

     blankstate:.InitialState &
     all arrow,zero in blankstate
         zero = 0 &
         rule in rules &
        ( arrow in rule.extneeds | arrow in rule.extgives )
     end &
     Putw (1,Nl,
   'Now indicate initial state by changing number of each arrow',
   ' which is enabled',Nl) &
     Modw (1,blankstate) &
     initstate = blankstate

{-------------------------------------------------------------}
subr InString (prompt:<S, reply:>S) iff
{input a string}
    Putw(1,Nl,prompt,Nl) &
    Getw (1,reply)
```

```
subr InSlist (prompt:<S, reply:>ExtArrows) iff
    work:.ExtArrows &
    work := Nil &
    Putw(1,Nl,prompt, Nl) &
    Modw (1,work) &
    reply = work

{================ Make new process by modifying initial state
                              of existing process================}

subr ModState (old:<S, new:<S) iff
{find old process in file,
 new is a name not already in file,
 modify the initial state of old process and insert new process
 with new name in file}

   ~ one x x = 1 & ProcessF(new,_,_,_,_) end &
                                          {not a duplicate name}
   one roles, state, rules, cchoice
      ProcessF (old, roles ,state, rules, cchoice)
   end &
   {modify state}
   newstate:.InitialState &
   newstate := state &
   Putw (1,Nl,
     'Now REVISE initial state by changing number of each arrow',
                ' which is enabled',Nl) &
   Modw (1,newstate) &
   process:>ExtProcess &
   process = (new, roles, newstate, rules, cchoice) &
   Insert (ProcessF, process)

{--------------- Revise file by DELETE, MODIFY ----------------}

subr ReviseP (i:<I) iff
     plist :> list ExtProcess &
     all p in plist
         ProcessF (p) end &
     Rewind (ProcessF) &
     Truncf (ProcessF) &
     Revise1 (plist)

PatchWindow :< Window =      {so you can make modifications to
                              a process specification}
               Frame(Single,Vert(Fill(1,75,
    ('Modify this process now, then press <F2>',Nil)),
                                  Pane(1,20,75),Nil))
```

```
local subr Revise1 (plist:<list ExtProcess) iff
     Print (N1, plist.h.name) &
     Popw (6,50,Dpopattr, ['Keep', 'Modify', 'Delete'], option) &
     case option:[0..3] of
     0 => Put (ProcessF,plist.h);
     1 => extprocess := plist.h &
               {-- optional modifications -- }
          Openw (4,1,0,PatchWindow) &
          Modw (1,extprocess) &
          Closew (4) &
          Put (ProcessF,extprocess) ;
     else
          true  {Delete by omiting Put}
     end &
     if plist.t <> Nil then
        Revise1 (plist.t)
     end

{--------------- Make Arbiter Specification ------------------}

subr ArbiterSpec (n:<I) iff
{creates spec for an n-Arbiter, and writes it on ProcessF}
   Append (Ltos (n),'-Arbiter',name) &
   Print (name) &
   role:.ExtRole &
   initial:.InitialState &
   rules :. ExtRules &
   role := Nil &
   initial := ('critical',1),Nil &
   rules := Nil &
   Interface (1, n, role, initial, rules) &
   Insert (ProcessF, (name,role, initial,rules,Nil))

proc Interface (i:<I, n:<I, role:.ExtRole, initial:.InitialState,
                            rules :. ExtRules) iff
   {add the required ports, arrows and rules for this interface}
   Chtos ("a"-1+i, ur) &                  {user request port}
   Chtos ("p"-1+i, ua) &                  {user acknowlege}
   role := (ur,In),(ua,Out),role &
   Append (ur,'1',a1) &                   {4 arrows}
   Append (ur,'2',a2) &
   Append (ur,'3',a3) &
   Append (ur,'4',a4) &
   initial := (a1,1),(a2,0),(a3,0),(a4,0),initial &
   rules := ((ur,0),Plain, (a1,Nil),(a2,Nil),Nil),
            ((ua,0),Choice,(a2,'critical',Nil),(a3,Nil),Nil),
            ((ur,1),Plain, (a3,Nil),(a4,'critical',Nil),Nil),
            ((ua,1),Plain, (a4,Nil),(a1,Nil),Nil),
            rules &
   if i < n then
      Interface (i+1, n, role, initial, rules)
   end
```

```
{declarations for DI. Network verification program, modules
 netverify and arrowcheck

      Lin Jensen
      January 26, 1990
      Modified:  November 1990 for BEHAVIOR states
      Modified:  December 1990 back to execution states
      Frequent changes in terms of actual array sizes.

This program verifies safety of a network of Delay Insensitive
processes, and liveness to the extent of progress requirements of
the specification.

 Declarations imported from process_t :
 ------------ -------- ---- ------------
Mark  = Plain | Loop | Choice
          (starts of Generators are marked to indicate looping or
             choice cuts -- used for hashing and forking of alg.)

NumberArrows = [0..255]
                  (state may have 0,1 or more arrows of same type)
Role = In | Out | Unused
Prime = Noprime | Oneprime | Prime2 | Prime3        (needs 2 bits)

(----- the following constants give the size of enumerated types
       which are used as array indexes.  This is needed by Setup
       and netverify to initialize arrays )
       (10 Pi's)

       (sizes of arrays as currently defined in netverify_t)
       (appropriate for up to 10 DME.
                    for buffers, 5 ports, 6 arrows is enough)
Nports:<I =   21
Narrows:<I =  41
Nprocess:<I = 11
(------------------------- STATES ------------------------------)

     ((global) state is changing (io variable).  It is the vector
      of states of the individual processes.  Logically this is a
      flexible array (of flexible arrays).
      HOWEVER, to enable efficient code generation and GARBAGE
      collection, it should be FIXED size, hence we will impose
      limits on both number of arrows per process and number of
      processes, both of which can be changed by recompiling.)


(The generic names of processes, used to index arrays)

Pr = Pm | P1 | P2 | P3 | P4 | P5 | P6
        | P7 | P8 | P9 | P10
    (   | P11 | P12)
    (   | P13 | P14 | P15
```

```
        | P16 | P17 | P18 | P19 | P20
        | P21 | P22 | P23 | P24       )

(Spec is mirrored to Pm, implementation is up to n processes)

(Generic arrow names, used to index)

Arrow =( A0 | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 |
           A11 | A12
         | A13 | A14 | A15 | A16 | A17 | A18 | A19 | A20 | A21
         | A22 | A23 | A24
         | A25 | A26 | A27 | A28
         | A29 | A30 | A31 | A32 | A33 | A34 | A35 | A36 | A37 |
           A38 | A39 | A40 )
        [0..40]

Arrows = list Arrow
      (for needs, gives :  list seems more efficient than vector)

(NumberArrows = [0..255]
              -- state may have 0,1 or more arrows of same type)

(Active = [0..1])

(Nov 1, 1990 ====================   BEHAVIOR STATE  ========

      Need to record the last source event of the active arrows
      (oops, if n arrows are active, this means n source events.
      This data structure will have to be modified for BUFFERing
      situations) Here we will record the last sources of all
      arrows (initially it's INIT) for unambiguous backward
      searches of causality.

   In the TABLE only active arrows & sources will be recorded.

Thus, for each arrow, we need Inactive or Active (0/1), last
     source (Port, Prime) and to keep the already extravagant
     space contained, we will PACK this info. into one I.  This
     can be done in asembly language. In fact, need 32,4,2
     combinations => 5 + 2 + 1 = 8 bits)

Pstate = Arrow -> I
     (packed source(port, prime), NumberArrows; source only if
     unique. Process state is given by the number of arrows of
     each type)

State = Pr -> Pstate
                    (global state is the states of each process)

("Hash" table declarations moved to netverify )
(-------------------------  PROCESSES  ----------------------)
```

```
Port = INIT |
       Port1 | Port2 | Port3 | Port4 | Port5 | Port6 | Port7 |
       Port8 | Port9 | Port10 | Port11 | Port12
      Port13 | Port14 | Port15 | Port16
    | Port17 | Port18 | Port19 | Port20
   ( Port21 | Port22 | Port23 | Port24 | Port25 | Port26 )

(Role = In | Out | Unused)

Portrole = Port -> Role     (each port is always either in or out)

Portname = Port -> S
                              (external names for ports and arrows)

Arrowname = Arrow -> S

(Prime = Noprime | Oneprime | Prime2 | Prime3 | Prime4)

Action = (Port, Prime)        (contextually, semantically different
                                        actions at the same port are
                                            distinguished by primes)

(Mark  = Plain | Loop | Choice )
                                    (starts of Generators are marked to
                                        indicate looping or choice cuts
                                -- used for hashing and forking of alg.)

Rule = (act:Action, mark:Mark, needs:Arrows, gives:Arrows,
        tickles:Arrows)

(Rule for a port is which arrows the port needs to be enabled,
 and which  new arrows are given by the occurence of the action.
 The needs arrows are "used up".  In addition, one tickles arrow
 will be needed without being used up.)

Rules = list Rule

          (of processes a network is built.  A process has a name
     e.g. 'DME' a set of ports, a set of arrows, an initial state
          coded as arrows enabled, and a table of enabling rules.

        Each port is designated either In or Out.  To mirror the
        specification, the In/Out role of each port is reversed.)

Process =
          ( name:S, portname:Portname, arrowname:Arrowname,
            portrole:Portrole,
            initialstate:Pstate, rules:Rules,
            coupled:list list Action )

(for coupled choice, each list Action is a set of concurrently
enabled choice)
```

```
(---------- NETWORK ----------------------------------------)

Link = ((Pr,Port),(Pr,Port))
Links = list Link
            (Network is formed by linking output to input ports)

(NOTE:  Ports but not primes (Actions) are mentioned.  One
process may prime its actions and another may not. Since this is
D.I., only one port action at a time may be enabled, so the state
determines which actions correspond.
HOWEVER, for arrow checking searches, we must assume that all
       processes have been fleshed out so that primes correspond)


Processes = Pr -> Process

Work_f  = file Processes
                       (setup needs to avoid running out of space)

Production = list Action
                      (one production of spec (order is in rules))

Productions = list Production

Network = (processes:Processes, links:Links, prods:Productions)

NetFile = file Network          (Setup will dump Network on a file,
                                 thus its garbage will be released)

(----------- ENABLED EVENTS ---------------------------------)

Event     = (p:Pr, r:Rule)
                      (an event in a network is an output action
                       of some process, and we wish to carry its
                       rule along.  It is coupled by a network
                       link to some input event)

EventPair = (oute:Event, ine:Event)
Elist     = list EventPair
                             (for listing all the enabled events.
           Recall that the rules include marking for checkpoints)

(-----------   For external stacking of  choices -------------)

StackF = file (Elist, State)
   (must record choice to be taken (and other marked events),
               and current state)

YesNo = Yes | NO                    (for whether to go on or not)

(---------------- end of module netverify_t -------------------)
```

(top level code of delay insensitive network verification program

```
                    POM: The program
                       Lin Jensen
                    January 28, 1990

  module :       netverify
  used modules: process_t netverify_t arrowcheck setup

  procedures:  NetVerify :  Construct the net from processes
                            with links and start the verification

               Verify :   Proceed with verification, given
                          hashlist of (important, hence marked,)
                          states reached, continue until:

               Hash :     State has been reached for which all
                          enabled events are marked as loop or
                          choice "points". Either quit or
                          proceed, hashing this state. If
                          choices, fork and try them all.

               Move :     Make the moves in a list of enabled
                          events, i.e. "chunk at a time." Update
                          state as you go, arrow check of dotted
                          (non-causal) arrows.
}
{-----------------------------------------------------------------
Declarations for table of selected visited grammar states. This
applies equally well to behavior or execution states.  In the
case of execution states, the source of ambiguous arrows is left
as "unknown", coded as INIT      }

Tablesize :< I = 8                    {number of table entries - 1}

BehState = [0..Narrows] -> (I,I)
                                 {vector of Pr,Arrow,Source,Active}
                {must be    (I,I)  for behavior states}

Statestab = [0..TableSize] -> BehState
                        {for "hash table" states that have occured}
States = (empty:I, a:Statestab}

{-----------------------------------------------------------------}
```

```
subr NetVerify (title:<S) iff
{"Main program"  construct a network, and verify it}
   Print (Nl,'Network verfication for: ', title) &
{   Dates (date) &
   Print (Nl,'Today is ',date) &}
   Print (Nl,Nl,'Processes      ----- Ports -----') &
    one net NetF(net) end &
   PrintProc (Pm, net.processes) &   {identify all the processes}
    Rewind (Stack) & Truncf(Stack) &            {EMPTY THE STACK}
    initialstate:.State &
{  Make initial state, all 0's, then fetch actual process states}
   empty :> Pstate &
   Dupl (Narrows, 0:NumberArrows, empty) &
   Dupl (Nprocess, empty, initialstate) &
               {initialize hash table (array of states) -------}
   hash :. States &
   hash := (0, Dupl (TableSize+1,
              Dupl(Narrows+1,(0,0:NumberArrows))):Statestab) &
   FetchPstate (net, Pm, initialstate)   &
                                    {get actual initial states}
   Verify (initialstate, hash, net, Nil) &
                                {start, no spec production chosen}
   DoForks (hash, net) &
   Print (Nl, 'Verification suceeded! ')

{-------------Make the initial network state by combining the
             initial process states into a matrix -------------}

proc FetchPstate (net:<Network, pr:<Pr, state:.State) iff
   state(pr) := net.processes(pr).initialstate &
   if next = pr + 1 then
      FetchPstate (net, next, state)
   end

{------------------  DoForks ----------------------------------}

subr DoForks (hash:.States, net:<Network) iff
{continue verification from choice points on the Stack}

   if Access (Stack,(elist,state1)) then
      Delete (Stack) &
      state :. State &
      state := state1 &
      LoadProd (elist, net, prod) &
      Move (elist, state, net) &
      RemProd (prod, elist, newprod) &
      Verify (state, hash, net, newprod) &
      DoForks (hash, net)
   end

{ --------------------- VERIFY -------------------------------}
```

```
subr Verify (state :. State, hashlist:. States, net :< Network,
                              prod:<Production) iff
            (do the verification from this state)
   MovePlain (state, net, moves_done, prod, newprod) &
   if moves_done = Yes then
      Verify (state, hashlist, net, newprod)
   else                                (all enabled events are marked)
    (Hash, then find those which involve choice (at either port),
                         the rest are Loop checkpoints)
      Hash (state, hashlist, net, go_on, prod, anothernewprod) &
      if go_on = Yes then
         Verify (state, hashlist, net, anothernewprod)
    (else Done with this branch, since state found in hash table)
      end
   end

(----------------------- HASH ---------------------------------)

subr Hash (state :. State, hashlist :. States, net :< Network,
           go_on :> YesNo,
           oldprod:<Production, newprod:>Production) iff

(at this state, all the enabled events are marked indicating
 the start of a new production, either looping or choice.
   IF this state already in hashlist, we are done!
   ELSE put in hashlist and continue,
        FORKing for all the choices  )

   cstate :. BehState &
   cstate := Dupl(Narrows+1,(0,0:NumberArrows)):BehState &
   Compress(state, cstate) &       (Behav. state to compact form)
   Add2Hash (0, cstate, hashlist, size, go_on_h) &
                       (find Choice events (either the in or out))
   if go_on_h  = Yes then
      Print ('    -----> Hash # ',size,'  <---') &
      Enabled (state, net, allenabled) &
      choicelist :> Elist &
      all choicee in choicelist
          choicee in allenabled &
          choicee.oute.r.mark = Choice
      end &
      if Len(choicelist) <= 1 then
                                 (no choice, go on with all events)
                              (only one "choice" is also no choice)
         go_on = Yes &
         if oldprod = Nil then
            LoadProd (allenabled, net, nextprod) &
            Move (allenabled, state, net) &
            RemProd (nextprod, allenabled, newprod)
         else
            (--- make just the moves which are internal net ones
                    or are spec actions in the current production
```

```
                                                    e.g. coinitial )
            internal :>Elist &
            all inev in internal
                inev in allenabled &
              ( inev.oute.p <> Pm | inev.oute.r.act in oldprod ) &
              ( inev.ine.p <> Pm | inev.ine.r.act in oldprod )
            end &    (must be some events!)
            if internal = Nil then
               Print (Nl,
             'LIVENESS VIOLATION, CANNOT REACH END OF SPEC PROD')
               & false
            else
               Move (internal, state, net) &
               RemProd (oldprod, internal, newprod)
            end
          end
        else                    {============= CHOICES ============}
         if oldprod <> Nil then
            Print (Nl,
                 'ERROR OF CHOICE IN THE MIDDLE OF SPEC COMMAND') &
            false
         else
           {make list of Loop (non-choice) events}
           all loope in looplist
               loope in allenabled &
               ~ loope in choicelist
           end &
         {FORK.  Pick one choice pair and all Loop events for each
                                                          branch}
         {NOTE:  This is naive and needs refinement--------------}
           Print (Nl, '-------- START FORK ----') &
           Fork (choicelist, looplist, state, net) &
                                              {STACK all choices}
           go_on = NO &
           newprod = Nil
{            Print (Nl, '   ...choosing ') &
            Printevent (1,choicelist.h,net.processes) &
           picked = (choicelist.h,looplist) &
           LoadProd (picked, net, nextprod) &
           Move (picked, state, net) &
           RemProd (nextprod, picked, newprod)}
         end
        end      {of choices}
  {else DONE!  state was in hashlist}
   else
       go_on = NO &
       Print (Nl,
      '==== found current state in hashlist====== entry # ',size)
       & newprod = Nil
   end   {--- of Hash ----}

{ --------------------- FORK --------------------------------}
```

```
subr Fork (choicelist :< Elist, looplist :< Elist, state :< State,
           net :< Network) iff
{for each choice, this proc makes a copy of the state and then
proceeds }
   if choicelist <> Nil then
            Print (Nl, '   ...deferring... ') &
            Printevent (1,choicelist.h,net.processes) &
      Put (Stack, ((choicelist.h,looplist),state)) &    {to FILE}
      Fork (choicelist.t, looplist, state, net)
   end

{------------------------- MOVE ----------------------------}

subr MovePlain (state:.State, net:<Network, moves_done:>YesNo,
                oldprod:<Production, newprod:>Production) iff
{make all the plain moves enabled at this state}

   Enabled (state, net, allenabled) &  {calc. all enabled events}
                                        {find those which are Plain,
                                     i.e. don't start any new prod.}
   all eventpr in plainlist
       eventpr in allenabled &
       eventpr.oute.r.mark = Plain &
       eventpr.ine.r.mark = Plain
   end &
   if plainlist <> Nil then
      Print (Nl, '    ... moving') &
      Move (plainlist, state, net) &
      RemProd (oldprod, plainlist, newprod) &
      moves_done = Yes
   else
      newprod = oldprod &
      moves_done = NO
   end

subr Move (elist :< Elist, state :. State, net :< Network) iff
{make all these moves, updating state}
   if elist <> Nil then
      Print (Nl, '          event pair ') &
      Printevent (1, elist.h, net.processes) &
      UpdateState (state(elist.h.oute.p), elist.h.oute.r) &
      UpdateState (state(elist.h.ine.p),  elist.h.ine.r ) &
      Arrowcheck (elist.h, state, net) &
                                          {check ...> to in_event}
                                  {safety, and liveness if Spec(Pm)}
      Move (elist.t, state, net)
   end
{-------------------------------------------------------------}
proc RemProd (old:<Production, elist:<Elist, new:>Production) iff
{remove those actions in the event list which are spec actions}
   EtoA(elist,done) &
   SubActs (old, done, new)
```

```
proc LoadProd (elist:<Elist, net:<Network, prod:>Production) iff
(load the appropriate production)
   EtoA (elist, these) &
   if these = Nil then
      (no spec events, do not load any production)
      prod = Nil
   else
     if  one pprod
          pprod in net.prods &
          Contains (these, pprod)
      end then
      prod = pprod &
      Print (Nl, '-++++-> Starting spec command ') &
      Printprod (prod, net.processes(Pm).portname) &
      Print ( '  <-++++-')
     else
      Print (Nl,
      '+++-> ERROR, unable to find command containing these ',
             'spec. actions:',Nl,'      ')&
      Printprod (these, net.processes(Pm).portname) &
      Print (Nl,' --- among these commands:') &
      Printprods (net.prods, net.processes(Pm).portname) &
      prod = Nil
     end
   end


local proc EtoA (elist:<Elist,   done :>Production) iff
(condense event list to list of spec actions)
   all act in done
       ep in elist &
       (ep.oute.p = Pm & act = ep.oute.r.act
      | ep.ine.p  = Pm & act = ep.ine.r.act )
   end

pred Contains (these:<Production, prod:<Production) iff
(these is a subset of prod)
   these = Nil | these.h in prod & Contains (these.t,prod)


(------------ put in hash table -------------------------------)

local proc Compress (state:.State, cstate:. BehState) iff
(compress state into compact form, recording just active arrows)
   behlist :> list (I,I) &
   all prarrow, status in behlist
      state(pr,arrow) = status &
(      Unpack (state(pr,arrow),_,#,status) & )
      Is_Active (status) &
    ( prarrow = pr:I*256 + arrow )
      Pack2 (pr, arrow, prarrow)
   end &
```

```
   n = Len(behlist) &
   if n > Narrows + 1 then
          Print (Nl,'TOO MANY ACTIVE ARROWS FOR TABLE ',n)
   end &
   PutB (0, behlist, cstate)

local proc PutB (i:<I, blist:<list (I,I), cstate:.BehState) iff
(put list of active arrows into array for hash table use
                                                  "behavior state")
if blist <> Nil then
   cstate (i) := blist.h &
   PutB (i+1, blist.t, cstate)
end


proc Add2Hash (n:<I, state:<BehState, hash:.States, place:>I,
                                              go_on :> YesNo) iff
(add this state to hash table.
   Sets go_on = NO if the state is already there.  This ends the
            recursion. Fails with note if the hash table is full)

    if n > TableSize then
       Print (Nl, 'HASH TABLE FULL, CANNOT COMPLETELY VERIFY') &
       false
    elsif n >= hash.empty then                    (empty table entry)
       hash.a(n) := state &                      (add state to table)
       hash.empty := n + 1 &
       place = n + 1 &            (report position counting from 1)
       go_on = Yes

    elsif hash.a(n) <> state  then       (not there, keep looking)
       Add2Hash (n+1, state, hash, place, go_on)
    else
       place = n + 1 &
       go_on = NO
    end

local proc PrintProc (pr:<Pr, p:<Processes) iff
   Print (Nl, pr, '   ',p(pr).name, '    ') &
   PrintPorts (Port1, p(pr).portname) &
   if pn = pr +1  & p(pn).name <> 'NULL' then
      PrintProc (pn, p)
   end

local proc PrintPorts (p:<Port, pname:<Portname) iff
   Print (pname(p),',') &
   if pn = p + 1 & pname(pn) <> 'NC' then
      PrintPorts (pn, pname)
   end

(---------------- end of module netverify --------------------)
```

```
{
  ================ module arrowcheck ====================

    part of Delay Insensitive Network Verification
    -- Restricted Process Theory --
    by Lin Jensen
      February 21, 1990
      November  4, 1990 -- entirely new version, combined
                           safety and liveness
      December  4, 1990 -- remove print statements
                        -- develop alternate past branching
                           for exec. states
      December  9, 1990 -- optimization for simple arrows,
                           code at end
      December 13, 1990 -- code for multiple arrows: PC,
                           buffer situations

    type declarations are in process_t and netverify_t
    used by netverify


===  extra declarations: ========================================}

AA = (act:Action, active:NumberArrows)      {presets, to compare}
Alist = list AA

Space30:<S = '                                '

{-------------------------------------------------------------}

proc Arrowcheck (event:<EventPair, state:<State, net:<Network) iff
{check the dashed arrows leading to the input event of this pair
 ALL AT ONCE by finding causal preset and comparing with non-causal.
 If input is in MIRROR, then liveness is also checked  }

{---- state has been updated by event ----}

   pr = event.ine.p &
   act =
       Lookup_act (event.ine.r.act, net.processes(pr).portname) &
 (                 {--- first try simple case---}
   Safe   (event.ine.r, state(pr), noncause) &
                                        {find non-causal preconds}
   Direct_c (event, state, net, direct) &
   direct = noncause
   {---------------< optimize, if simple check, we are done <---}
     |
   {---------------> if that failed, the complete check :       }
   Search (event, state, net, causal) &   {find causal preconds.}
   Presets (event.ine.r, net.processes(pr), state(pr),
            noncausal) &                  {find non-causal "    }
                                                 {compare them}
```

```
   if pr = Pm then
   (------ Pm is SPECIFICATION, check safety and LIVENESS ------)
      if (causal = noncausal)
         Live (noncausal, causal, event.ine.r, net.processes(Pm))
       then
         true
(       & Print (Nl,Space30,
                  'Safety and liveness verified for ',pr,', ',act))
      else
         Print (Nl,'SAFETY/LIVENESS VIOLATION (arrows) at ',
                                                  pr,', ',act) &
         false
      end
   else
   (------- Pi is network component, check saftey only ---------)
   Print (Nl,'noncausal = ') &
            Printpreset (noncausal, net.processes(pr).portname) &
   Print (Nl,'causal =    ') &
               Printpreset (causal, net.processes(pr).portname) &
      if Safety (noncausal, causal) then
         true
(     & Print (Nl,Space30,'Safety verified for ',pr,', ',act) )
      else
         Print (Nl,'SAFETY VIOLATION (arrows) at ',pr,', ',act) &
         false
      end
   end
)

(-------------------------------------------------------------)

Item = Pr,Action
Items = list Item
(list of actions visited while searching, to stop infinite loops)

local proc Search (event:<EventPair, state:<State, net:<Network,
             causal:>list Alist) iff

(search for the causal preconditions of the input event, in its
 process. If process = Pm (spec), liveness checking is on, only
 find direct causals. state records source actions of unambiguous
 arrows. In case of non-determinism, there will be branching on
 alternate pasts of an execution state.  Thus a list of sets of
 preceeding events will be returned.  The proper starting point
 for this list is a list containing precisely the empty set (not
 an empty list).  This is given as (Nil,Nil) )

   pr = event.ine.p &                    (searching in this process)
   Causal (pr, 0, event.oute.p, event.oute.r.needs, state, net,
           Nil (visit), (Nil,Nil) (sofar), causal)
                     (find enabling events by transitive closure)
```

```
{-------------------------------------------------------------}
{      Explicit Depth First Search, with alternate sources      }
{-------------------------------------------------------------}

local proc Causal (pr:<Pr,
            activecount:<NumberArrows,  {count active arrows met}
             {target:<Event{Out},} pt:<Pr, needs:<list Arrow,
             state:<State, net:<Network, visit :< Items,
             sofar:<list Alist, sources:>list Alist{Out}) iff

   { each set in sources is a set of actions in pr
               causally preceeding target,
               directly preceeding if pr is spec (Pm),
               that is, no intervening Pm actions are allowed,
               on account of liveness }

 if needs = Nil then
   sources = sofar                          {search complete}
 else
                                  {search for causes of this arrow}
  Unpack (state(pt)(needs.h), port,prime,active) &
  moreactive = activecount + active &
                                {accumulate count of active arrows,
                                   if any, met on backward search }
  events :> list Event &
                {will find one or list of conflicting pre evvents}
  if port <> INIT then                      {arrow has one source!}
    Unique (pt,port,prime, net, unique) &
                                  {matched to unique output event}
    events = unique,Nil              {so Depth will not backtrack}
  else
      { === Pre can BACKTRACK on alternate sources of arrow === }
    all event in events
      Pre (event{out}, pt, needs.h, state, net)
                                          {back up over one arrow}
    end
  end &
     {===== explicit backtracking in Depth if alternate events =}
   Depth (events, pr, moreactive, state, net, visit, (Nil),
          source) &

   { = breadth =  search for causes of remaining arrows }
   Causal (pr, activecount, pt, needs.t, state, net, visit,
           Blend (source,sofar), sources)
           {combine sources of each arrow}
 end

{-------------------------------------------------------------}
```

```
local proc Depth (events:<list Event,
             pr:<Pr, {searching in this pr}
             activecount:<NumberArrows,
             state:<State, net:<Network, visit :< Items,
             sofar:<list Alist, sources:>list Alist{Out}) iff

{search by depth, with BRANCHING on alternate source Events}
  if events = Nil then
     if sofar = Nil then
        sources = Nil,Nil
     else
        sources = sofar
     end
  else
   event = events.h &
   px = event.p &
   item:>Item &
   item = px,event.r.act &
  if  item in visit  then                          {no repeated events}
        sources = sofar
  else
   if px = pr then                                              {found one}
      treasure = ((event.r.act,activecount),Nil):Alist &
                      {report the action & count of active arrows}
      Print(Nl,'Found event ',
           Lookup_act(event.r.act, net.processes(pr).portname)) &
    { Print ('  visited ') & Pvisit(visit,net.processes)  & }
      if pr = Pm then
         source = treasure,Nil
                             {not past spec event when liveness on}
      else  {keep looking further back}
         Causal (pr, activecount, px, event.r.needs, state, net,
                 (item,visit),(treasure,Nil), source)
                               {record this act since it's in pr}
           {transitive closure}
           { = depth = }
      end
   else
      Causal (pr, activecount, px, event.r.needs, state, net,
              (item, visit),(Nil,Nil), sourc) &
                                          {transitive closure}
      if sourc = Nil,Nil then
         source = Nil
      else
         source = sourc
      end
  end &
  Depth (events.t, pr, activecount, state, net,
          visit, Append (sofar, source), sources)
                   {DIFFERENT source lists represent Non-Determ.}
   end
end
```

```
{-------------------------------------------------------------}

local pred Pre (event:>Event{Out), pt:<Pr, {target process}
                arrow:<Arrow,          {arrow target act needs}
                state:<State, net: Network) iff
   {the input action linked to source is directly arrow connected
    to target in its process, working backwards}

{one} inevent in net.processes(pt).rules &
                                        { <==== BACKTRACK HERE  }
      arrow in inevent.gives &  {which is a source of this arrow}
      inevent.act = port,prime &
      if net.processes(pt).portrole(port) = In then
         sin = port,prime &              {source input act in pt}
         Linked ((pt,sin), oute, net.links) &
         oute = pr,source &  {gives the process of the out event}
         rule in net.processes(pr).rules &
         source = rule.act &
         event = pr,rule
      else
      {------- DONT LINK IF sin is actually OUTPUT port!!!!!----}
         event = pt,inevent   { in fact it was an output event
                             from an output ordering of a process}
      end

{-------------------------------------------------------------}

local proc Unique (pt:<Pr,port:<Port,prime:<Prime, {input action}
                   net:<Network,           {network connections}
                   event:>Event(out) )       {linked output act}
                                       iff
{ alternate to Pre when there is no ambiguity in the arrow, so
the source (input) action is known, stored in state(arrow)     }

    sin = port,prime &                   {source input act in pt}
    if net.processes(pt).portrole(port) = In then
       one event
         Linked ((pt,sin), oute, net.links) &
         oute = pr,source &  {gives the process of the out event}
         rule in net.processes(pr).rules &
         source = rule.act &
         event = pr,rule
       end
    else
      {------- DONT LINK IF sin is actually OUTPUT port!!!!!----}
       one inrule
           inrule in net.processes(pt).rules &
           inrule.act = sin
       end &
       event = pt,inrule   { in fact it was an output event from
                             an output ordering of a process}
    end
```

```
{-------------------------------------------------------------}

local pred Linked (ine:<(Pr,Action), oute:>(Pr,Action),
                   links:<Links) iff
   {the events are linked --
                        ine backing up to oute that tickled it}
   {in fact this should be unique, but Trilogy can't know this}

   ine = inp,inport,inprime &

   outprime = inprime &
                      {special rule for DME -- & precooked in gen}
{=== in general, we need an additional data structure to record
     the correspondences between actions and actions ======}

   (outp, outport),(inp, inport) in links &
   oute  = outp,outport,outprime

{-------------------------------------------------------------}

Primes :< Prime -> S = ['','''','''''','''''''']

{ {============ print list of visited actions for debugging ====}
local proc Pvisit (visit:<Items , pros:<Processes) iff
   if visit <> Nil then
      visit.h = pr,(port,prime) &
      Print ( ' ',pr,':',pros(pr).portname(port),Primes(prime)) &
      Pvisit(visit.t,pros)
   end
   ========================}

local proc Lookup_act (action:<Action, names:<Portname, label:>S)
                       iff
   {look up the external port name for printing}
   action = port,prime &
   label = Append (names(port),Primes(prime))

{-------------------------------------------------------------}

{ Determining non-causal presets of execution state, with
  branching for alternate sources of arrows.
 Checked out for interesting arbiter event, Dec. 4, 1990  }

local proc Presets (event:<Rule, process:<Process, state:<Pstate,
                    presets:>list Alist) iff
{find alternate presets of this event in it's own process
 - non-causal}
   all preset in presets
       Direct (event.needs, process, state, Nil, pres) &
       Sort   (pres,preset)
   end
```

```
{------------------------------------------------ ------------}

local pred Direct (needs:<list Arrow, process:<Process,
                state:<Pstate, sofar:<Alist,  preset:>Alist)   iff
{predicate for direct non-causal preconditions for each arrow to
an event}
   if needs = Nil then
      preset = sofar
   else
      Unpack(state(needs.h),port,prime,active) &
      if port <> INIT then                    {GOT IT}
         act = port,prime
      else
  (one)  event in process.rules &      ( <==== BACKTRACK HERE  )
         needs.h in event.gives &
                                  {which is a source of this arrow}
         act = event.act
      end &
      Direct (needs.t, process, state, ((act,active):AA,sofar),
              preset)
                                                  {rest of preset}
   end

local proc Sort (messy:<Alist, nice:>Alist) iff
   all x in nice                                   {Trilogy sorts}
       x in messy
   end

{-------------------------------------------------------------------}

local proc Live (noncausal:<list Alist, causal:<list Alist,
                 currentevent:<Rule, spec:<Process) iff
   Print (Nl,'noncausal = ') &
   Printpreset (noncausal, spec.portname) &
   Print (Nl,'causal =    ') &
   Printpreset (causal, spec.portname) &
(   noncausal = causal           {exact agreement, assume sorted}
 |  Live2 (noncausal, causal,  currentevent, spec) )

local proc Printpreset (presets:<list Alist, ports:<Portname) iff
   if presets <> Nil then
      Print ('(') &
      Printoneset (presets.h, ports) &
      Print (')(') &
      Printpreset (presets.t, ports)
   else
      Print ('Nil')
   end

local proc Printoneset (preset:<Alist, ports:<Portname) iff
   if preset <> Nil then
      preset.h.act = port, prime &
```

```
      Print ('(', ports(port)) &
      Pprime (1,prime) &
      Print (',', preset.h.active,'),') &
      Printoneset (preset.t,ports)
   end

local proc Live2 (noncausal:<list Alist, causal:<list Alist,
                  currentevent:<Rule, spec:<Process) iff

{liveness requires that each non-causal preset has a causal
 equivalent.  That is, either the preset is exactly in causal, or
 is there with perhaps less active arrows.  In that case, we must
 additionally be able to imply that there are enough causal
 active arrows in the state. This can be shown with another
 preset (in causal) which has enough active arrows for an action
 which is a possible source for the same non-causal arrow. }

 if noncausal <> Nil then
    ( noncausal.h in causal
     |
      one other        {some noncausal preset agrees in actions }
         other in causal &
         SameActs (other, noncausal.h, Nil, checkmore) &
         CoverArrows (checkmore, causal, currentevent, spec )
               {.. and liveness can be inferred for mult. arrows}
      end
    ) &
    Live2 (noncausal.t, causal, currentevent, spec)
                                  {rest of presets}
 end

local proc SameActs (less:<Alist, more:<Alist,
                     sofar:<Alist, problems:>Alist) iff
                                          {list active problems}
{both lists have the same actions, but less may have less active
 arrows}
   if less <> Nil then
                               {acts will be sorted in same order}
      less.h.act = more.h.act &
      less.h.active <= more.h.active &
      if  less.h.active = more.h.active then
         newprobs = sofar
      else
         newprobs = more.h, sofar
      end &
      SameActs (less.t, more.t, newprobs, problems)
   else
      more = Nil &
           {can't have extra noncausal = safety violation 7/4/91}
      problems = sofar
   end
```

```
local proc CoverArrows (problems:<Alist, causal:<list Alist,
                        currentevent:<Rule, spec:<Process) iff

(this procedure will be invoked only when there is both non-
 determinism and multiple arrows in a state.  When this happens,
 we are not guarenteed to find representatives of all the active
 arrows in a particular causal chain )
(The problem arrows can be accounted for by other alternate
  sources. if (e,n) is in problems, that means that e is in a
  causal preset, but with less active arrows than n.  We must
  find, in some preset of causal, an event f (maybe even e
  itself) with n active arrows, where f is an alternate source of
  the same non-causal arrow as e, with target of the current
  event)
    if problems <> Nil then
       problems.h = e,n &
       one preset
           preset in causal &                    (in some preset .. )
           f,n in preset &     (there is an event with n actives)
           erule in spec.rules &
           erule.act = e &
           arrow in erule.gives &
                                (..which gives the same non-causal
                                     arrow as the problem event e)
           frule in spec.rules &
           frule.act = f &
           arrow in frule.gives &
           arrow in currentevent.needs
                                (and this arrow leads to the curr.e)
       end &
       CoverArrows(problems.t, causal, currentevent, spec)
                                                   (any more probs)
    end
(-------------------------------------------------------------)

local proc Safety (noncausal:<list Alist, causal:<list Alist) iff
   (safety only of noncausal arrows)
   ( --- all causal presets support a noncausal (safety)
         requirement set -- )

  if noncausal <> Nil then
      one safety
          safety in causal &
          Subset (noncausal.h, safety)
      end &
      Safety (noncausal.t, causal)
   end

local proc Subset (x:<Alist, y:<Alist) iff
(x is a subset of y in terms of actions,
                       with less than or equal active arrows)
   ~ one e e in x &  ~ e in y  end
```

```
(-------------------------------------------------------------)

local proc Blend (x:<list Alist, y:<list Alist, z:> list Alist) iff
(cartesian product of sets of sets, can blend alternate causes
 with others)
      (ASSUMPTION, neither x nor y is Nil (empty),
             they may contain the empty set only,
             but not the empty set and something else)
  all sources in z
     p in x &
     q in y &
     Union (p,q,sources)
  end


local proc Union (p:<Alist, q:<Alist, u:>Alist) iff
(set union:  p U q = u)
   all e in u
      e in p | e in q
   end

(=============================================================
       The following code optimizes checking of simple cases
 =============================================================)

local proc Safe (inev:<Rule, state:<Pstate, noncausal:>Alist) iff
   (find the set of non-causal preconditions of an input action
    in a process, from the event's needs arrows, and their last
    source, ONLY if there is no conflict )

   all act in noncausal
       act :> AA &
       arrow in inev.needs &
       Unpack (state(arrow), port, prime,active) &
(       port <> INIT &                (ignore INIT as trivial case))
       act = (port,prime),active
   end &
   ~ one ambiguous                  (all arrows have def. sources!)
       ambiguous in noncausal &
       ambiguous.act = INIT,Noprime
     end

(-------------------------------------------------------------)

local proc Direct_c (event:<EventPair, state:<State,
                     net:<Network, causal:>Alist) iff
(--- optimization --- find direct causal arrows in a coprocess )
   pr = event.ine.p &
   copr = event.oute.p &
   all source in causal
       Pre_c (pr, source, event.oute, state, net)
   end
```

```
local pred Pre_c (pr:<Pr, source:>AA(Out), target:<Event(Out),
                  state:<State, net:<Network) iff
   (the input action linked to source is directly arrow connected
    to target in its process, working backwards -- simple arrows
    only )

   pt = target.p  &                                  (target's process)
   arrow in target.r.needs &                    (all its arrow needs)
   Unpack (state(pt)(arrow),port, prime, active) &
   port <> INIT &            (Can't use simple check for conflict,
           but such arrows may not be needed for safety checking)

                                  (get source, fail on active arrow)
      if net.processes(pt).portrole(port) = In then
         sin = port,prime &                (source input act in pt)
         Linked ((pt,sin), oute, net.links) &
         oute = pr,act &
                        (can fail if pr is bound, else gives an x)
         source = act, active
      else
      (----- DONT LINK IF sin is actually OUTPUT port!!!!!------)
         source = (port,prime),active &
         pr = pt
      end

(---------------- end of module arrowcheck ---------------------)
```

```
(part of D.I. verification.  Make a network from DB of processes

   Lin Jensen
   January 28, 1990
   November 16, 1990:  Version with "tickle" arrows
   November 30         Added SetupDME to automate setup of ring
                       of DME
   December  9, 1990   Use Packing to record source of
                       unambiguous arrows, INIT is now just a
                       code for ambiguous arrow (see arrowheck)

     module :       setup
     used modules:  process_t netverify_t ProcessF windows
     Used by:       netverify
-------------------------------------------------------------)

Flex   = [0..] -> S     (used by Popw)

       (------------- get the processes and links -------------)
subr  Setup (x:<I) iff
   Setup1 (x) &
   Setup2 (x)

subr SetupDME (size:<I) iff
(set up ring of DME elements)
    SetupDME1 (size) &
    SetupDME2 (size)

subr SetBuf (size:<I) iff
(set up a buffer of size single buffers)
    SetBuf1 (size) &
    SetBuf2 (size)

local subr Setup1 (x:<I) iff
   (ask for process names, find them in the database ProcessF)
   all pro in allpro ProcessF (pro) end &
   all name in prnames (name,_,_,_,_) in allpro end &
   Putw (1,
  'Pick processes for network from window. (ESC if not there)') &
(=== initialize process array to null processes:--)
   one extnul extnul in allpro & extnul.name = 'NULL' end &
   FixForm (P1, extnul, nul) &
   Print (N1,'Back from FixForm') &
   prlist :. Processes &
   Dupl (Nprocess, nul, prlist) &
   Print (N1,'Going to  AskProcess') &
   AskProcess ('Specification', Pm, allpro, prnames, prlist) &
   Print (N1,'Back from AskProcess') &
   Rewind (WorkFile) &
   Truncf (WorkFile) &
   Put (WorkFile, prlist)
```

```
local subr Setup2 (x:<I) iff
   Print (Nl, 'starting SEtup2') &
   Access (WorkFile, prarray) &
   Print (Nl,'going to AskLinks') &
   AskLinks  (prarray, links) &                    (then ask for links)
                     (ask for spec productions (sets of actions))
   AskProds (prarray(Pm), prods) &

   net:>Network &
   net = prarray, links, prods &
   (empty NetF and put this in it)
   Rewind (NetF) &
   Truncf (NetF) &
   Put    (NetF, net)
                  (output to a file so that garbage produced
                  by setup will be discarded --  also useful
                  so verificaton can be done several times, and
                  network can be saved (by DOS) for later)

local subr SetBuf1 (size:<I) iff
(set up the network of a ring of DME elements)
(=== initialize process array to null processes:--)
   one extnul ProcessF(extnul) & extnul.name = 'NULL' end &
   one nbuf
       ProcessF(nbuf) &
       nbuf.name = Append(Ltos(size),'-Buffer')
   end &
   one dme ProcessF(dme) & dme.name = 'Buffer' end &
   FixForm (P1, extnul, nul) &
   Print (Nl,'Back from FixForm') &
   prlist :. Processes &
   Dupl (Nprocess, nul, prlist) &
   prlist(Pm) := FixForm (Pm,nbuf) &
   AddBuf (P1, size, FixForm(P1,dme), prlist) &
                                        (as many Buf's as needed)
   Rewind (WorkFile) &
   Truncf (WorkFile) &
   Put (WorkFile, prlist)

local proc AddBuf (pr:<Pr, num_to_do:<I, dme:<Process,
                                          array:.Processes) iff
 (add a dme process to array of processes)
     array(pr) := dme &
     if num_to_do - 1 > 0 then
        AddBuf (pr+1, num_to_do-1,dme, array)
     end

local subr SetBuf2 (size:<I) iff
   Print (Nl, 'starting SEtup2') &
   Access (WorkFile, prarray) &
   Print (Nl,'going to AskLinks') &
   BufLinks  (size, prarray, links) &
```

```
                                                    (then ask for links)
                        (ask for spec productions (sets of actions))
   AskProds (prarray(Pm), prods) &

   net:>Network &
   net = prarray, links, prods &
   (empty NetF and put this in it)
   Rewind (NetF) &
   Truncf (NetF) &
   Put    (NetF, net)
      (output to a file so that garbage produced by setup will be
       discarded --  also useful so verificaton can be done
       several times, and network can be saved (by DOS) for
       later)

local subr SetupDME1 (size:<I) iff
(set up the network of a ring of DME elements)
(=== initialize process array to null processes:--)
   one extnul ProcessF(extnul) & extnul.name = 'NULL' end &
   one arb
      ProcessF(arb) &
      arb.name = Append(Ltos(size),'-Arbiter')
   end &
   one ldme ProcessF(ldme) & ldme.name = 'LDME' end &
   one dme ProcessF(dme) & dme.name = 'DME' end &
   FixForm (P1, extnul, nul) &
   Print (Nl,'Back from FixForm') &
   prlist :. Processes &
   Dupl (Nprocess, nul, prlist) &
   prlist(Pm) := FixForm (Pm,arb) &
   prlist(P1) := FixForm (P1,ldme) &
   AddDME (P2, size-1, FixForm(P2,dme), prlist) &
                                          (as many DME's as needed)
   Rewind (WorkFile) &
   Truncf (WorkFile) &
   Put (WorkFile, prlist)

local proc AddDME (pr:<Pr, num_to_do:<I, dme:<Process,
                    array:.Processes) iff
                        (add a dme process to array of processes)
     array(pr) := dme &
     if num_to_do - 1 > 0 then
        AddDME (pr+1, num_to_do-1,dme, array)
     end

local subr SetupDME2 (size:<I) iff
   Print (Nl, 'starting SEtup2') &
   Access (WorkFile, prarray) &
   Print (Nl,'going to AskLinks') &
   DMELinks  (size, prarray, links) &
                                                    (then ask for links)
                        (ask for spec productions (sets of actions))
```

```
   AskProds (prarray(Pm), prods) &

   net:>Network &
   net = prarray, links, prods &
   {empty NetF and put this in it}
   Rewind (NetF) &
   Truncf (NetF) &
   Put    (NetF, net)
      {output to a file so that garbage produced by setup will be
       discarded --  also useful so verificaton can be done
       several times, and network can be saved (by DOS) for
       later}

{so you can patch in modifications temporarily to a process
specification}

PatchWindow :< Window =
               Frame(Single,Vert(Fill(1,75,
('You may modify this process now before including it in network:
<F2>',Nil)),
                                 Pane(1,20,75),Nil))

local subr AskProcess (blurb:<S, pr:<Pr,
                       allpro:<list ExtProcess, prnames:<list S,
                       prlist:.Processes ) iff
{for sucessive pocesses, use popup window to display the
available process names, add the entire process to the list}

   Putw (1,Nl,'Pick ',blurb, ' process ',pr, ': ') &
{   Rewind (ProcessF) &  }
   Len(prnames) <= 23 &
                      {max for Popw  fail for now, later provide}
                      {              ---------------------------}
   namearr = (Len(prnames),prnames):Flex &
   Popw (0,50, Dpopattr, namearr , pick) &
   pick < Len(prnames) &                    {fail if ESC pressed}
   Putw (1,namearr(pick)) &
  { Skip (ProcessF, pick) &
                        {since names & file are kept ordered, the
                      value returned by Popw = # records to skip}
   Access (ProcessF, extprocess) &           {get from the file}
   -------------- Skip appears not to be working for DB }
   one extproc
       extproc in allpro & extproc.name = namearr(pick)
   end &
{  extprocess:.ExtProcess &                      ---- commented out
   extprocess := extproc &
 {-- optional modifications -- }
   Openw (4,1,0,PatchWindow) &
   Modw (1,extprocess) &
   Closew (4) &
   FixForm (pr, extprocess, process) &    {internal form}
```

```
)  FixForm (pr, extproc, process) &                    (internal form)
   prlist(pr) := process &
   if nextp = pr+1 then
      AskProcess ('Network', nextp, allpro, prnames, prlist)
                                                    (tail RECURSION)
   end

Wlink:<Window =                          (window for asking for links)
      Frame (Single, Vert ( Fill (3,40,(
             'For each output port in the network',
             'you will select the linked (P, port)',
             ' Pr. OUT  Port    Pr. IN  Port',
       Nil) ), Pane(1,6,40), Nil))

Portlist = list (Pr, Port, S)

local subr AskLinks (prarray:<Processes, links:>Links) iff
    outports :> Portlist &
    all pr, port, name in outports
        prarray(pr).portrole(port) = Out &
        prarray(pr).portname(port) = name
    end &
    Openw (3, 1,5, Wlink) &
    AskLink (prarray, outports, Nil, links) &
    Closew (3)

local subr AskLink (prarray:<Processes, outports:<Portlist,
                                  sofar:<Links, links:>Links) iff
   if outports = Nil then
       links = sofar
    else
       outports.h = po,p,name &
       Putw (1, '  ',po,'     ', name) &
       Popw (6,45, Dpopattr, ['Spec - Pm', '1  -   P1','2  -   P2',
                 '3  -   P3', '4  -   P4', '5  -   P5','6  -   P6',
                 '7  -   P7', '8  -   P8', '9  -   P9',
                 '0  -  P10', '-  -  P11',
                 '= -  P12','[ - P13',']  - P14'],pii) &
       {ESC = no link}
      if pii = Nprocess then
          AskLink (prarray, outports.t,  sofar, links)
      else
       pi = pii:Pr &
       {find input ports for that process (pi)}
       connected :> list Port &
       all prt in connected
                        (these have been connected => unavailable)
           (_,pi,prt) in sofar
       end &
       inports :> list (S,Port) &
       all name,port in inports
           prarray(pi).portrole(port) = In &
```

```
            ~ port in connected &
                                        {can't link to already linked}
            prarray(pi).portname(port) = name
        end &
        len = Len(inports):[0..Nports] &
        case len of
           0 => Print (Nl, 'NO INPUT PORTS for process ',pi) &
                false ;        {NO ports!}
           1 => defaultname,inport = inports.h &
                Putw(1, '    ',pi,'     ', defaultname, Nl) ;
           else
                inname :> list S &
                all name in inname   name,_ in inports end &
                Popw (8, 60, Dpopattr, (len,inname):Flex, choice) &
        {        if choice = len then  {ESC}
                   AskLink (prarray, outports, sofar, links)
                                                        {REPEAT}
                else  }
        {pick one}
                   inporta = (len, inports):[0..]->(S,Port) &
                   inporta(choice) = choicename, inport &
                   Putw(1, '    ',pi,'     ', choicename, Nl)
{                 end  {if} }
           end   {case} &
        {inport}
        link = ((po,p),(pi,inport)):Link &
        AskLink (prarray, outports.t, (link, sofar), links)
      end   {if not ESC of Process}
    end    {if not Nil}

local proc DMELinks (size:<I, prarray:<Processes, links:>Links)
          iff
    outports :> Portlist &
    all pr, port, name in outports
        prarray(pr).portrole(port) = Out &
        prarray(pr).portname(port) = name
    end &
    DMELink (size, prarray, outports, Nil, links)

local proc DMELink (size:<I, prarray:<Processes,
              outports:<Portlist, sofar:<Links, links:>Links) iff

   if outports = Nil then
      links = sofar
   else                    {assign input to first outport on list}
      ipr:>Pr &
      outports.h = opr, oport, oname &
      if opr = Pm then
         ipr = oname(0)-"a"+1:Pr &
         one iport
             prarray(ipr).portname(iport) = 'ur'
         end
```

```
      else                        (a DME, out ports are ua, la, & rr)
         if oname = 'ua' then
            ipr = Pm &
            iname = Chtos ("o"+opr:I) &
            one iport
                 prarray(ipr).portname(iport) = iname
            end
         elsif oname = 'la' then
            if opr:I < size then
               ipr = opr + 1
            else
               ipr = P1
            end &
            one iport
                 prarray(ipr).portname(iport) = 'ra'
            end

         elsif oname = 'rr' then
            if opr <> P1 then
               ipr = opr - 1
            else
               ipr = size:Pr
            end &
            one iport
                 prarray(ipr).portname(iport) = 'lr'
            end
         else
            Print(Nl,'Snafu with ',opr,oname) &
            false
         end
      end &
      link = ((opr,oport),(ipr, iport)):Link &
      DMELink (size, prarray, outports.t, (link,sofar), links)
   end

local proc BufLinks (size:<I, prarray:<Processes, links:>Links) iff
    outports :> Portlist &
    all pr, port, name in outports
        prarray(pr).portrole(port) = Out &
        prarray(pr).portname(port) = name
    end &
    BufLink (size, prarray, outports, Nil, links)

local proc BufLink (size:<I, prarray:<Processes,
              outports:<Portlist, sofar:<Links, links:>Links) iff
(assign the links for a multiple buffer made of size simple
buffers)
   if outports = Nil then
      links = sofar
   else                    (assign input to first outport on list)
      OneBlink (size:Pr, prarray, outports.h, link) &
      BufLink (size, prarray, outports.t, (link,sofar), links)
```

```
   end

local proc OneBlink (plast:<Pr, prarray:<Processes,
                     outport:<(Pr,Port,S), link:>Link) iff
      outport = opr, oport, oname &
      Print(Nl,opr,':',oname,' = ') &
      {=== special cases:  Pm: a, d connect to P1 a, d
                       size:Pr: b, c connect to Pm  b, c
           normal case.    Pi: c, b   connect to Pi+1  a, d   ==}
      if opr = Pm then
         if oname = 'a' then
            ipr = P1
         else {'b'}
            ipr = plast
         end &
         iname = oname
      elsif opr = plast & Print('**') & oname = 'c' then
         ipr = Pm & iname = oname
      elsif  opr = P1 & oname = 'd'  then
         ipr = Pm & iname = oname
      else
         if oname ='c' then
            ipr = opr+1 & iname = 'a'   {next buffer}
         elsif oname = 'd' then
            ipr = opr-1 & iname = 'b'   {previous}
         else
            Print ('SNAFU') & false
         end
      end &
      {find internal name}
      Print (ipr, ':',iname) &
            one iport
                 prarray(ipr).portname(iport) = iname
            end &
      link = ((opr,oport),(ipr, iport)):Link

{---------- checked out for C-element Jan 30,1990 L.J.--------}
local proc FixForm (index:<Pr, external:<ExtProcess,
                    internal:>Process) iff
 {convert process from external to internal representation}
   Print (Nl,'fixform for ',external.name) &
                    {convert port list to array of names & roles}
   ports:.Portname &
   Dupl (Nports,'NC',ports) &
   roles:.Portrole &
   Dupl (Nports,Unused,roles) &
   if external.role <> Nil then
      FixPort (external.role, Portl, ports, roles)
   end &
   if index=Pm then                                {this is the SPEC}
      Mirror (Portl, roles)                              {mirror it}
   end &
```

```
   {convert arrows, initstate}
   arrows:.Arrowname &            {store external names of arrows}

   initstate:.Pstate &
   Dupl (Narrows,'@',arrows) &
   Dupl (Narrows,0:I{NumberArrows}, initstate) &
   if external.initial <> Nil then
      FixArrow (external.initial, {A}0, arrows, initstate)
   end &
   {convert rules}
   rules :> Rules &
   all rule in rules
       extrule in external.extrules &
       ConvertRule (index, ports, arrows, extrule, rule)
   end &
   (-- figure out which are the simple arrows and record their
       sources --}
   Sources ({A}0, arrows, rules, initstate) &
   {convert coupled}
   all couple in coupled
       extcouple in external.extcoupled &
       ConvertCouple (ports, extcouple, couple)
   end &
   internal = (external.name, ports, arrows, roles, initstate,
               rules, coupled){:Process}


local proc FixPort (plist:<ExtRole, pt:<Port, ports:.Portname,
                    roles:.Portrole) iff
      plist.h = name, role &
      ports(pt) := name &
      roles(pt) := role &
      if plist.t <> Nil then       {could fail if too many ports}
         FixPort (plist.t, pt+1, ports, roles)
      end

local proc FixArrow (alist:<InitialState, ar:<Arrow,
arrows:.Arrowname,
state:.Pstate)   iff
      alist.h = name, count &
      arrows(ar) := name &
      state (ar) := count &    {(INIT,NoPrime,number_active)}
      if alist.t <> Nil then
         FixArrow (alist.t,
                   ar+1,   {-> could fail if too many arrows}
                   arrows, state)
      end

proc Mirror (port:<Port, roles:.Portrole) iff
{the mirror of a process is the process, with the roles In & Out
 reversed}
   case roles(port) of
```

```
      In  => roles(port) := Out;
      Out => roles(port) := In ;
      else true
   end &
   if next = port+1 then Mirror (next, roles) end

local proc Sources (arrow:<Arrow, arrows:<Arrowname, rules:<Rules,
              state:.Pstate) iff
  (if the source of this arrow is unambiguous, pack it into state,
   then do same for next arrow)
   if arrows(arrow) <>'@' then      (remaining arrow names unused)
      all s in sources
         rule in rules &
         arrow in rule.gives &      (event which gives this arrow)
         s = rule.act
      end &
      if sources = unique,Nil then  (exactly one source)
         unique = port,prime &
         active = state(arrow) &
         state(arrow) := Pack (port,prime,active)  (.. record it)
      end &
      if next = (arrow + 1):Arrow then
         Sources (next, arrows, rules, state)
      end
   end


local pred  ConvertRule (pr:<Pr, ports:<Portname,
             arrows:<Arrowname, extrule:<ExtRule, rule:>Rule) iff
(--- convert one rule from external to internal form ---- )
                           (act)
   act:>Action  &
       extrule.extact = name,i &
       name = ports(port) & prime = i:Prime &
       act = port,prime &
   if pr <> Pm & extrule.mark = Loop then
      mark = Plain                  (only mark SPEC loop events)
   else
      mark = extrule.mark
   end &
                           (needs)
   needs:>Arrows &
   CodeArrows (arrows, extrule.extneeds, needs) &
                           (gives)
   gives:>Arrows &
   CodeArrows (arrows, extrule.extgives, gives) &
                           (tickles)
   tickle:>Arrows &
   CodeArrows (arrows, extrule.exttickles, tickle) &
   rule = (act, mark, needs, gives, tickle)
```

```
local proc CodeArrows (arrows:<Arrowname,
                       external:<ExtArrows, {list S of arrows}
                       internal:>Arrows)  iff
{look up index i.e. internal name for each arrow. Note order
 doesn't matter}
   all arrow in internal
       name in external &
       name = arrows(arrow)
   end

{-------------------------------------------------------------}

local pred ConvertCouple (ports:<Portname, ext:<list ExtAct,
                          couple:>list Action) iff
 {convert coupled events to internal form}
    if ext = Nil then
       couple = Nil
    else
       ConvertCouple (ports, ext.t, some) &
       ext.h = name,i &
       name = ports(port) &                    {find port in table}
       prime = i:Prime &
       couple = (port,prime),some
    end

{-------------------------------------------------------------}

local proc AskProds (process:<Process, prods:>Productions) iff
{figure out the the grammar productions of the specification
 using 'coupled'}

  {figure out the starting actions}
  plainacts:>Production &
                {plain actions, which can be attached to a prod.}
  all act in plainacts
      rule in process.rules &
      rule.mark = Plain &
      rule.act = act
  end &
  startact:>Production &
  all act in startact
      rule in process.rules &
      rule.mark <> Plain &
      rule.act = act
  end &
       {process.coupled has lists of co-initial starting actions}
  Ask1Prod (startact, plainacts, process, Nil, prods)
```

```
local proc Ask1Prod (startact:<Production, plainacts:<Production,
                     process:<Process, sofar:<Productions,
                     prods:>Productions) iff
    if startact = Nil then
       prods = sofar
       & Print (Nl, 'COMMANDS ARE :', Nl)
       & Printprods (prods, process.portname)
    else
   (given starting actions, figure out what plain actions follow)
(--  when more than one starting action, these will be listed --)
       if one clist
          clist:>list Action &
          clist in process.coupled &
          startact.h in clist
          end
       then
          start = clist
       else
          start = startact.h,Nil
       end &

       Follows (start, prod, process.rules, plainacts) &
       SubActs (startact, prod, restofacts) &
       Ask1Prod (restofacts, plainacts, process, (prod, sofar),
                 prods)
    end

local proc Follows (start:<Production, prod:>Production,
                    rules:<Rules, acts:<Production) iff
   all x in next
     x in acts &
     x in start &
     r1:>Rule &
     r1 in rules &
     r1.act in start &
     r2 in rules &
     r2.mark = Plain &
     a in r1.gives &
     a in r2.needs &
     r2.act = x
   end &
   if next = Nil then
      prod = start
   else
      Follows (Append(start,next), prod, rules,
               SubActs(acts,next))
   end

(---------------- end of module setup --------------------)
```

```
{Lin Jensen
 Delay insensitive network verification
 Jan - Mar. 1990

---------  find ENABLED states (used also by netverify) ---------
```

module enabled uses process_t netverify_t used by setup arrowcheck netverify

```
==========================================================================}

proc Enabled (state:<State, net:<Network, elist:>Elist) iff

{for the given state, produces a list of pairs:
  (rule for enabled output action, rule for linked input action)
     the calling proc will split into lists of Plain, Loop or
     Choice actions}

   EnableP (state, net, Pm, Nil, elist)        {for each process}


proc EnableP (state:<State, net:<Network, process:<Pr,
              sofar:<Elist, elist:>Elist) iff
{add enabled events for this process}

     EnableR (state, net, process,
              net.processes(process).rules,
                                        {Rules for this process}
              Nil,                     {event pairs so far in p}
              more) &
     if nextp = process + 1 then
        EnableP (state, net, nextp, Append(sofar,more), elist)
     else
        Append (sofar, more, elist)
     end

{-------------------------------------------------------------------}

proc EnableR (state:<State, net:<Network, process:<Pr,
              {build a list of enabled events for this process, }
              prule:<Rules,       { by looking through its rules}
              sofar:<Elist,       { events found so far this pr.}
              allthisp:>Elist) iff

   if prule = Nil then
      allthisp = sofar      {done for this process.}
   else
      if    {it's an output rule & }
         rule = prule.h &
         rule.act = outport,_ &
         net.processes(process).portrole (outport) = Out &
```

```
            (the event is enabled by state)
         Subset (rule.needs, state(process))  &
         Tickled (rule.tickles, state(process))  (tickle enabled)
      then
                                  (find linked input action, rule)
         if
             one                (FAILS if no linked action enabled)

(ASSUMPTION made here that only one rule for the inport
  will be applicable at any one state (and there better be one).
  The only wierd exception I can think of would be a process that
  could intrepret its input as a request for either ONE soda or
  TWO bananas)

                  inprocess,inrule
                  (process,outport),(inprocess,inport) in net.links &
                  inrule in net.processes(inprocess).rules &
                  inrule.act = inport,_ &
                  Subset (inrule.needs, state(inprocess)) &
                  Tickled (inrule.tickles, state(inprocess))
              end
         then

            (add to list)
            newevent :> EventPair &
            newevent = (process,rule),(inprocess,inrule) &
            EnableR (state, net, process, prule.t,
                     (newevent,sofar), allthisp)
         else                                        (SAFETY VIOLATION)
            Print (Nl, 'SAFETY VIOLATION at ') &
            Pel (1, (process, rule), net.processes) &
            allthisp = Nil &
            false                          (==== want to FAIL =====)
         end
      else
         EnableR (state, net, process, prule.t, sofar, allthisp)
      end
   end

(----------------  STATE TESTING & MANIPULATION ----------------)

local proc Subset (needs:<Arrows, state:<Pstate) iff
  (needs (for an action) is a subset of the current state of its
   process.  Needs is a list of arrows, state a multiset vector)

   if needs <> Nil then
      Is_Active (state(needs.h)) &   (each arrow exists in state)
      Subset (needs.t, state)
   end
```

```
proc Tickled (tickle:<Arrows, state:<Pstate) iff
(tickle enabling exists if it is required)
    if tickle <> Nil then        (suceed if no tickling required)
       one a                    (or find one active tickle arrow)
           a in tickle &
           Is_Active (state(a))
       end
    end
(-------------------------------------------------------------)

proc UpdateState (state:.Pstate, rule :<Rule) iff
(for the process involved, update its state vector by the
 rule of the event we are just moving over.
             state is a multiset (vector of counts) )

(there is a peculiar enabling of rr in the DME, that a
 request will be made for the token if either interface
 wants service, WITHOUT 'using up' the service request
 arrow.  The arrow just 'grazes' the event (rr).  This is
 implemented by putting the alternate arrows in tickles.  It
 is important for safety checking to not loose sight of the
 true source of the arrow AND, we hope, not important to
 record that it touched rr. SO tickle arrows don't affect
 updated state)

   SubA (rule.needs, state) &
                              (remove arrows used up by the action)
   AddA (rule.act, rule.gives, state)
                                 (add arrows given by the action)

local proc SubA (needs:<Arrows, state:.Pstate) iff
            (needs (for an action) is a subset of the current
             state of its process.
             Subtract these arrows from the state.
             Needs is a list of arrows, state a multiset vector)
   if needs <> Nil then
         state(needs.h) := state(needs.h) - 1
               (each arrow exists in state when Subset was called
                     (we know event is enabled) -- make inactive)
      &
      SubA (needs.t, state)
   end

local proc AddA (act:<Action, gives:<Arrows, state:.Pstate) iff
      ( Add each arrow in gives to the state.
        gives is a list of arrows, state a (multi)set vector
        each arrow indexes an entry of state, which is packed
        (port,prime, active).  )
   if gives <> Nil then
      state(gives.h) := state(gives.h) + 1 &   (exec. state)
      AddA (act, gives.t, state)
   end
```

```
{-------------------------------------------------------------}

proc SubActs (x:<Production, y:<Production, diff:>Production) iff
(set difference of lists of actions)

all z in diff
    z in x & ~ z in y
end

{-------------------------------------------------------------}

proc Printevent (pid:<I, e :< EventPair, net:<Processes) iff

(Print an event pair, using external names. (format it nicely) )

   Pe1 (pid, e.oute, net) &
   Print (' = ') &
   Pe1 (pid, e.ine, net)

proc Pe1 (pid:<I, e :< Event, net:<Processes) iff
(print each event in the pair)
     e.r.act = port,prime &
     name = net(e.p).portname(port) &
     Print ( ' ', e.p,', ',  name) &
     Pprime (pid, prime)

proc Pprime (pid:<I, p:<Prime) iff
   (print a number of primes corresponding to
                          prime field of an event)
  if p <> Noprime then
     Print ('''') &
     Pprime (pid, p-1)
  end

proc Printprods (prods:<Productions, names:<Portname) iff
(print all the productions in the list)
    if prods <> Nil then
       Print (Nl, '     ') &
       Printprod (prods.h, names) &
       Printprods (prods.t, names)
    end

proc Printprod (prod:<Production, names:<Portname) iff
(print one 'production', a list of actions (port, prime) )
   if prod <> Nil then
      prod.h = port, prime &
      Print ('  ', names(port)) &
      Pprime (1,prime) &
      Printprod (prod.t, names)
   end

{--------------- end of module enabled -------------------}
```

```
{module packing.
  Pack and Unpack arrow info in state.  For each (Pr,Arrow),
           there will be a word type I,
           but logically it holds (Port, Prime, NumberArrows),
           number of bits  =      ( 5,     2,     8    )

Author:  Lin Jensen
Date:    November 1, 1990
For:     D.I. Network Verification,  netverify and all.

These will be implemented in assembly language  }

proc Pack (port:<Port, prime:<Prime, active:<NumberArrows, p:>I)
   iff
     extern

proc Unpack(p:<I, port:>Port, prime:>Prime, active:>NumberArrows)
   iff
     extern

proc Is_Active (pack:<I(NumberArrows)) iff extern
   {true iff the arrow status is active => active > 0}

proc Pack2 (process:<Pr, arrow:<Arrow, p:>I) iff extern
   {packs 2 bytes in word, equivalent to but faster than
      p = process*256 + arrow }

{======== Here is the assembly language code:

 ;assembly language code for the trilogy module packing.
 ;To be used by  NetVerify.
;       Author:  Lin Jensen
;       Date:    November 1, 1990
;                  p:  logically it holds (Port, Prime, Active),
;                  number of bits  =      ( 5,     2,     8    )

dgroup  group   data
data    segment word public 'DATA'
data    ends


_code   segment byte
assume  cs:_code
assume  ds:dgroup

;proc Pack (port:<Port, prime:<Prime, active:<Active, p:>I) iff
  public        Pack
```

```
Pack    proc    far
        push    bp      ;save bp for Trilogy
        mov     bp,sp   ;new value of bp
;[bp+6] contains the 16 bit value of port,      only lsb signif.
;[bp+8]    "    "       "       "    prime              "
;[bp+10]   "    "       "       "    active             "
;[bp+12]   "    "       "   pointer to p   (packed info)

        mov     ah,[bp+6]       ;port
        shl     ah,1
        shl     ah,1
        or      ah,[bp+8]       ;prime
        mov     al,[bp+10]      ;active, assume either 0 or 1
        mov     bx,[bp+12]      ;get pointer to p
        mov     [bx],ax         ;store packed value there

        mov     ax,1    ;return sucess
        pop     bp      ;restore bp
        ret             ;do not cut the arguments off the stack
 Pack    endp

;proc Unpack (p:<I, port:>Port, prime:>Prime, active:>Active) iff
  public        Unpack

Unpack  proc    far
        push    bp      ;save bp for Trilogy
        mov     bp,sp   ;new value of bp
;[bp+6] contains the 16 bit value of p     (packed info)
;[bp+8]    "    "       "   pointer to port
;[bp+10]   "    "       "       "      prime
;[bp+12]   "    "       "   pointer to active

        mov     ax,[bp+6]       ;packed value
        mov     dh,0            ;msb  of all output params = 0
        mov     dl,al           ;copy NumberArrows part
        mov     bx,[bp+12]      ; point to active
        mov     [bx],dx         ;  & save
        mov     dl,ah
        and     dl,03h          ;mask prime
        mov     bx,[bp+10]      ; point to prime
        mov     [bx],dx         ;  & save
        mov     dl,ah           ; get uppermost 6 bits = port
        shr     dl,1
        shr     dl,1
        mov     bx,[bp+8]       ;point to port
        mov     [bx],dx         ; save last 5 bits

        mov     ax,1    ;return success
        pop     bp      ;restore bp
        ret             ;do not cut the arguments off the stack
Unpack  endp
```

```
;proc Is_Active (pack:<NumberArrows) iff extern
;   (true iff the arrow status is active = 1)

  public        Is_Active

Is_Active       proc    far
        push    bp      ;save bp for Trilogy
        mov     bp,sp   ;new value of bp
;[bp+6] contains the 16 bit value of p     (packed info)
;       the lsb is number of arrows.
;   sucess if > 0 (above)
;   failure if  0 arrows active

        mov     ax,[bp+6]    ;return sucess or failure
        mov     ah,0         ; disragard any packed info in msb
        cmp     al,0         ; depending on active arrows = 0
        jz      decided         ;return the 0 as failure
          mov   al,1            ;return success
decided:
        pop     bp      ;restore bp
        ret             ;do not cut the arguments off the stack
Is_Active       endp


;proc Pack2 (process:<I, arrow:<I, p:>I) iff
;  (pack two bytes into a word, process in msb, arrow in lsb)
  public        Pack2

Pack2   proc    far
        push    bp      ;save bp for Trilogy
        mov     bp,sp   ;new value of bp
;[bp+6] contains the 16 bit value of process    only lsb signif.
;[bp+8]    "    "       "       "    arrow                "
;[bp+10]   "    "       "   pointer to p   (packed info)

        mov     ah,[bp+6]       ;process
        mov     al,[bp+8]       ;arrow
        mov     bx,[bp+10]      ;get pointer to p
        mov     [bx],ax         ;store packed value there

        mov     ax,1    ;return success
        pop     bp      ;restore bp
        ret             ;do not cut the arguments off the stack
 Pack2  endp

_code   ends
        end
}
{--------- ------ end of module packing --------------------}

{================ END OF PROGRAM LISTING ==================}
```

INDEX