USING PARTIAL-ORDER SEMANTICS TO AVOID THE STATE EXPLOSION PROBLEM IN ASYNCHRONOUS SYSTEMS

David K. Probst and Hon F. Li
Department of Computer Science
Concordia University
1455 de Maisonneuve Blvd. West
Montreal, Quebec Canada H3G 1M8

ABSTRACT

We avoid state explosion in model checking of delay-insensitive VLSI systems by not using states. Systems are networks of communicating finite-state nonsequential processes with well-behaved nondeterministic choice. A specification strategy based on partial orders allows precise description of the branching and recurrence structure of processes. Process behaviors are modelled by pomsets, but (discrete) sets of pomsets with implicit branching structure are replaced by pomtrees, which have finite presentations by (automaton-like) behavior machines. The latter distinguish both concurrency and branching points, and define a finite recurrence structure. Safety and liveness checking are integrated. In contrast to state methods, our methods do not require enumeration or recording of states. We avoid separate consideration of execution sequences that do not differ in their partial order, and ensure termination by recording only a small number of system loop cutpoints -- in the form of system behavior states. In spite of the name, behavior states are not states.

Keywords delay-insensitive system, model checking, state explosion, partial-order semantics, branching point, recurrence structure, behavior machine, behavior state.

1. Introduction

There is considerable interest in asynchronous hardware systems, fueled by concerns about clock distribution and component composition in clocked systems [1,3,6,8]. Clearly, there is considerable conceptual overlap between asynchronous hardware systems and asynchronous distributed systems. Here, we focus on hardware systems, although we make our verification assumptions clear, as a first step towards extending the work to other application areas. Most theoretical asynchronous systems research is based on a formal representation strategy that underlies efforts in both verification and synthesis, and can have a major impact on efficiency (for example, on the time needed for verification, or on the size of synthesized objects). In delay-insensitive VLSI systems, system correctness is independent of delays in both asynchronous circuit components and transmission media. Both are specified as asynchronous processes. Delay-insensitive systems are modelled as networks of processes that communicate by direct contact. Since all communication is asynchronous and unbuffered, only local protocols are available to eliminate undesirable inputs. We avoid state explosion in model checking of delay-insensitive VLSI systems by not using states. Our specification strategy allows precise description of the branching and recurrence structure of processes. Processes are modelled by pomtrees, which differ from (discrete) sets of pomsets in making implicit branching structure explicit. Pomtrees have finite presentations by (automaton-like) behavior machines that distinguish both concurrency and branching points, and define a

This research was supported by the Natural Sciences and Engineering Research Council of Canada under grants A3363, A0921 and MEF0040121.

finite recurrence structure. Our slogan is, combine true concurrency with true nondeterminism -- and a finite recurrence structure -- to achieve efficient algorithmic processability. Our model checking strategy is, evaluate graph predicates on a system pomtree rather than state predicates on a system state graph. A small set of loop cutpoints in a finite presentation of the system pomtree is discovered during model checking. The core novelty of our approach is that (i) causality is checked directly, and (ii) behavior states (which are not states) are used only for termination.

2. Abstract specification of asynchronous processes

Abstract specifications define sets of externally-visible infinite computational behaviors, as well as branching points and a finite recurrence structure. Our behaviors are called complete to emphasize that they correspond to some maximal safe use of the process by an environment. In event structure terms, a complete behavior is a maximal conflict-free set of events. A process P has a set of input ports I and a set of output ports O. A process action is a (port, token) pair -- in VLSI, tokens correspond to signal (voltage) transitions. tI (tO) is the set of input (output) actions, and $\Sigma = tI \cup tO$ is the set of process actions. An event at an input or output port performs a process action defined by the value of the token received or sent. Since the focus in this paper is on systems with cleanly defined control states that are verified separately, tokens are essentially colorless. P's input actions are under the exclusive control of P's environment. P's output actions are under the exclusive control of P. This asymmetry of control is central to the model.

Safety properties (invariance properties) specify what the process is allowed to do; they also specify what the environment is allowed to do. A safety violation is an occurrence of a proscribed input or output event (in automata-theoretic terms, an undefined transition). After an input safety violation, a process becomes undefined. Liveness properties (inevitability properties) specify what the process is required to do; in our model, they do not specify what the environment is required to do. A liveness violation is a nonoccurrence of a prescribed output event. We group liveness properties into two camps, viz., (1) those related to progress (that is, response), and (2) those related to fairness of conflict resolution. Symmetric (asymmetric) specification of safety (liveness) properties has an interesting structuring effect on model checking (the examination of a special closed network of processes): although safety violations can show up in any process, liveness violations can only show up in the specification.

2.1. Primitive notions

At the level of (infinite) pomtrees, there are two primitive notions: (1) complete behavior (maximal conflict-free set of events), that is, any maximal infinite path through the pomtree, and (2) nondeterministic choice between mutually exclusive sets of process actions, by either process or environment, that is, selection of a particular pomtree branch. Complete behaviors are abstractions of infinite executions of the process that (i) correspond to some maximal safe use, and (ii) contain only necessary temporal precedences between external (interface) events. Concurrent events have no specified temporal relationship. A behavior "contains" any execution state that arises in any execution abstracted by the behavior. Execution state is affected by the performance of process actions in the usual way. P is input nondeterminate when P's environment can choose; P is output nondeterminate when P can choose.

At the level of (finite) behavior machines, there is a third primitive notion: recurrence (looping) in behavior space. Actually, the major precondition for the application of our techniques is the existence of a clean, finite recurrence structure. Behavior machines describe how commands (socket-extended finite pomsets) define transitions between selected pairs of behavior states. The essential use of behavior states is to identify cutpoints in loops of computational behaviors. A behavior state contains all the information in an execution state plus some additional information about how events in the past are temporally related to events in the future. A behavior state is not a state in the sense that recording system execution states is not sufficient to discover a finite presentation of the system pomtree.

2.2. Pomsets and pomtrees

Pomsets and pomset operations have been studied extensively [4]. A labelled partial order (lpo) is a 4-tuple (V, Σ, Γ, μ) consisting of (i) a countable set V of events in a computational behavior, (ii) a finite set Σ of process actions, (iii) a partial order Γ on V that expresses the necessary temporal precedences among the events in V, and (iv) a labelling function $\mu : V \rightarrow \Sigma$ mapping each event $v \in V$ to the process action $\sigma \in \Sigma$ it performs. In our model, events at the same port in a given behavior are linearly ordered (concurrent events must be at distinct ports). Since every process has an initial state in which no events have occurred, and since the partial order Γ expresses action enabling (causal dependency among events), Γ must be well-founded (axiom of finite causes). Formally, a pomset (partially ordered multiset) is the isomorphism class of an lpo, denoted $[V, \Sigma, \Gamma, \mu]$.

Let p and q be pomsets, and let all partial orders Γ be written as "<". We say that q is a ρ-prefix of p when q is obtainable from p by deleting a subset of the events of p, provided that if event u is deleted and $u < v$, then v is also deleted. We say that α is a π-prefix of p when α is a finite ρ-prefix of p. $\pi(p)$ is the set of π-prefixes of p. We say that q is an augment of p when q differs from p only in its partial order, which must be a superset of that of p. If P is a set of pomsets, then $\pi(P)$ is $\cup \pi(p)$, $p \in P$. Let $p = [V, \Gamma, \Sigma, \mu]$ be a pomset and let $V' \subseteq V$. The projection onto V' is $p' = [V', \Gamma', \Sigma, \mu']$, where Γ' and μ' are the restrictions of Γ and μ to V'.

We introduce the notion of prefix envelope. Let p be a pomset. ${}^{\circ}p$ is the set of action labels of initial events of p, that is, the set of $\mu(v)$ of v in p such that $\nexists u$ in p with $u < v$. If $\alpha \in \pi(p)$, then $p - \alpha$ is the projection obtained by deleting the events in α from p. The envelope of α in p, denoted $E_p(\alpha)$, is ${}^{\circ}(p - \alpha)$. Define $in_p(\alpha) = E_p(\alpha) \cap tI$ and $out_p(\alpha) = E_p(\alpha) \cap tO$. $E_p(\alpha)$ is the set of process actions that are concurrently enabled in behavior p after the events of partial execution α. We sketch the notion of branch point. Consider the following skeleton presentation of a simple finite pomtree. Let α, β_1 and β_2 be finite pomsets, with ${}^{\circ}\beta_1 \cap {}^{\circ}\beta_2 = \{\ \}$. Lay down pomset α. Now, both concatenate β_1 with set Γ_1 of (α, β_1) precedences, and concatenate β_2 with set Γ_2 of (α, β_2) precedences. The pomtree contains one copy each of α, β_1 and β_2. β_1 and β_2 are the two branches. Let p and q be the two (maximal) paths through the pomtree. By construction, $\alpha \in \pi(p)$, $\pi(q)$ is a maximal (under prefix ordering) common prefix of p and q. Also by construction, $E_p(\alpha) \cap E_q(\alpha) = \{\ \}$. We say that α is a branch point. If $E_p(\alpha)$, $E_q(\alpha) \subseteq tI$ (tO), then α is an input (output) branch point. Pomtrees are like computation trees except that arcs are maximal determinate behavior segments, and vertices are input or output branch points.

2.3. Determinate processes

Fig. 1 shows a representation of a complete computational behavior p. Process P = {p} could be a C-element together with an unconnected wire. The use of pluses and minuses (from rising and falling signal transitions) is partly redundant; once the initial voltage level of a port has been fixed by a reset transition, each new transition at that port is the opposite of the preceding transition. Output port names have been underlined. The complete behavior is obtained by concatenating infinitely many copies of this figure, superimposing adjacent (a^+, b^+, d^+) triples. The figure is intended to illustrate both maximal safe use and necessary temporal precedence. For example, process P is a multithreaded process (it has two threads); the environment may elect to use one thread and not the other. A process with a single thread has essentially only one use. Necessary temporal precedence is explained below.

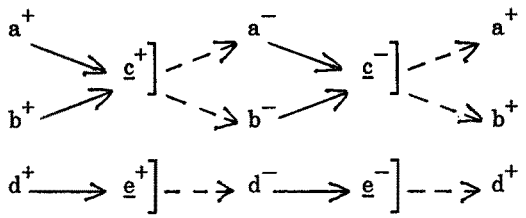

Fig. 1 Representation of a complete computational behavior.

The complete behavior precisely describes when the process (environment) is allowed to perform output (input) actions. In all processes, as illustrated in Fig. 1, the behavior partial order Γ in $p = [V, \Sigma, \Gamma, \mu]$ is the transitive closure Ω^+ of a nontransitive successor relation $\Omega = N \cup \Xi$, where the output protocol N is a relation from input events to output events, and the input protocol Ξ is a relation from output events to input events. In model checking, we use the fact that N (Ξ) is the causal (noncausal) part of Ω, by asymmetry of control. In short, the process (environment) enforces N (Ξ). We also say that the actions in $out_p(\alpha)$ [$in_p(\alpha)$] are causally [noncausally] enabled at α. The semantics of successor arrows is as follows. The process may perform an output action when all of its solid-arrow predecessor events have occurred. The environment may perform an input action when all of its dashed-arrow predecessor events have occurred. Violations of the input (output) protocol are called failures (errors) [6]. After a failure, a process may behave arbitrarily. This means that system pomtrees of (closed) networks of processes are undefined if there is a failure in any component process. Brackets are explained momentarily.

2.4. Delay insensitivity

In general, well-behavedness conditions are motivated by restrictions that allow asynchronous processes to be used as components of delay-insensitive systems [6]. Here, we present a minimal set of assumptions that are used in the model checking approach of this paper. Assumptions that guarantee delay insensitivity are distinguished from more general assumptions -- not shown in this paper -- that guarantee the finite-state character of asynchronous processes, and the bounded-size encoding of behavior states. These general assumptions are what guarantee the existence of a clean, finite recurrence structure in an asynchronous hardware or software system.

Well-behavedness conditions

Rule 1 There is no autoconcurrency. Formally, any two events at the same port in $p \in P$ are separated in Ω by at least one event at some other port.

Rule 2 There is no specified successor relationship either between two input events or between two output events. Formally, each line in $p \in P$ consists of an infinite sequence of strictly alternating input and output events.

2.5. Progress requirements

We specify progress requirements in a behavior p by bracketing output events. After partial execution $\alpha \in \pi(p)$, a determinate process $P = \{p\}$ is required to perform all bracketed output actions in $out_p(\alpha)$; this subset is denoted $req_p(\alpha)$. These actions must be performed eventually, after some arbitrary but finite delay. To simplify model checking, from now on we only consider systems in which $req_p(\alpha) = out_p(\alpha)$; this defines restricted process theory. We capture this notion in a rule.

Rule 3 In the absence of branching, there is never a choice between performing and not performing an output action. Formally, if $p \in P$, then all output events in p are bracketed.

2.6. Nondeterminate processes

Branching in an asynchronous process arises in either of two ways. There may be free choice (in the Petri net sense) in the environment to apply one of several mutually exclusive access operators, and there may be arbiter choice in the process to respond to one of several concurrent requests to perform a mutually exclusive operation on a shared object. Pomtrees adequately model the branching structure arising from these two sources. The semantics at input (output) branch points is straightforward. An input branch point α defines $\mathbf{in}(\alpha) = \{in_j(\alpha) : j \in J\}$, where J indexes the finite input branching at α. This is the family of disjoint sets of input actions that are concurrently enabled at α, respectively, in each of the futures that branch from α. An output branch point α defines $\mathbf{out}(\alpha) = \{out_k(\alpha) : k \in K\}$, where K indexes the finite output branching at α. This is the family of disjoint sets of output actions that are concurrently enabled at α, respectively, in each of the futures that branch from α. The semantics at nonbranch points is equally straightforward. If α is within a determinate behavior segment, then there is precisely one enabled set. If α straddles a branch point, then the enabled sets are not disjoint.

Consider an output branch point α in restricted process theory. If P advances to α, then P is required to perform all actions in some out_k within finite time. Explanation is required for arbiter choice, concerning progress requirements at prefixes of α. Suppose that output action u is required at β and that, before P performs u, P advances to α in which u and v are disjunctively required; P may then perform v and not u. This is an acceptable confusion situation (in the Petri net sense): in an arbiter, a required output action may become disjunctively required after the receipt of new input.

2.7. Behavior machines

Behavior machines are finite presentations of pomtrees. A behavior machine consists of a set of selected behavior states, and a set of commands. A command can be applied in certain behavior states, and produces a new behavior state. Commands are essentially finite pomsets with additional machinery to define the nonsequential serial concatenation -- modulo branching -- of the command to the finite pomtree generated so far. One fact of model checking is that components of systems are subject to nonmaximal safe use, requiring the encoding of behavior states that were not originally selected; for this reason, we encode all behavior states at specification time. Behavior machines specify both safety and progress properties; fairness properties must be specified by a supplementary condition constraining output choice.

Formally, a behavior machine is a set S of selected behavior states (with a distinguished start state), and a set C of commands. The machine describes the possible state transitions (c takes s to t) produced by each command. For each $c \in C$, there is a transition relation $\Delta(c)$ on S, where $(s, t) \in \Delta(c)$ if conceptual execution of command c in state s produces state t.

Fig. 2 shows a behavior machine for a C-element. Each command has the form: behavior state label, socket-extended finite pomset, behavior state label. In constructing behaviors, a new command may be applied provided the command prelabel matches the postlabel of the previous command. Sockets (denoted $\circ$) help describe how future behavior follows past behavior. If there is a successor arrow $\circ \rightarrow u$, then $\circ$ is filled by some predecessor of u according to a well-defined rule. More precisely, if command c can follow partial execution α, then there is an injection (produced by simple labelling) from the set of sockets in c to the set of events in α that can still participate in (new) successor arrows. By convention, there is an imaginary initialization event $\bullet$ with postlabel 0. As a presentational device, we make the vertical placement of symbols in commands significant: a socket is always filled by an action that appears on the same horizontal line. For example, in Fig. 2, the unique socket $\circ$ can be filled by $\bullet$ (first application of the command) or by $\underline{c}^-$ (all subsequent applications of the command). Assume that the default vertical placement of $\bullet$ is "middle".

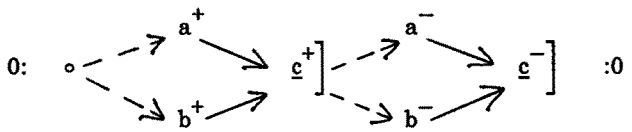

Fig. 2 Behavior machine for a C-element.

Fig. 3 shows a reduced behavior machine for a delay-insensitive arbiter. For conciseness of presentation, we have included several distinct behavior states under label 1. The abuse of notation is intentional. Label 1 in Fig. 3 is an equivalence class of behavior states. The two "commands" on the right in Fig. 3 are equivalence classes of commands. This is explained below. Each of two clients follows a four-cycle protocol, where $\langle A \rangle = \underline{c}^+ \rceil$ - -> a^- and $\langle B \rangle = \underline{d}^+ \rceil$ - -> b^- are the critical sections. A two-arrow socket that is always filled by • has been suppressed from command 1 (extreme left). As shown, sockets exist on three horizontal levels. The top ∘ (in command 2) is always filled by a^+ and the bottom ∘ (in command 3) is always filled by b^+. Either middle ∘ can be filled by •, a^- or b^-, perhaps redundantly.

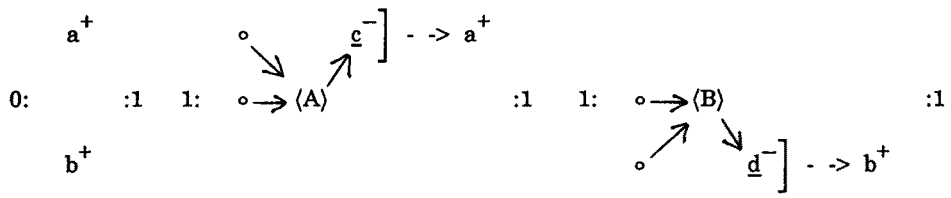

Fig. 3 Reduced behavior machine for a delay-insensitive arbiter.

The reduced behavior machine in Fig. 3 groups three distinct behavior states under label 1, all corresponding to the same execution state. The three may be distinguished as: (1.1) no critical-section entry has occurred (both middle sockets become redundant), (1.2) A's critical-section exit was the most recent (A's middle socket becomes redundant), and (1.3) B's critical-section exit was the most recent (B's middle socket becomes redundant). Behavior states 1.1, 1.2 and 1.3 are output branch points. Behavior states 1.2 and 1.3 form a complete set of loop cutpoints in the full behavior machine; such a set is called a dominator set [2,7]. The full behavior machine is easily obtained. It has four selected behavior states (0, 1.1, 1.2 and 1.3) and seven transitions -- but only five commands (distinct socket-extended pomsets). For example, behavior state 1.2 has a self-loop produced by command 2′ (command 2 minus its middle socket), and a transition to behavior state 1.3 produced by command 3.

The simplest way to encode behavior states is to use names of successor arrows. This is done by recognizing distinct arrows in a behavior machine, and assigning labels. For a finite-state determinate process P = {p}, the behavior state $s(\alpha)$ that corresponds to partial execution $\alpha \in \pi(p)$ is encoded as the set $\Omega(\alpha, p - \alpha)$ of successor arrows with source in α and target in $p - \alpha$. This encoding strategy works equally well for finite-state nondeterminate processes. For example, any behavior state of the arbiter can be encoded by three (small) integers representing the (virtual) successor arrows currently offered by the past to the future. Thus, behavior state 1.2 is encoded as: a^+ (b^+) is currently available to enable the next $\underline{c}^+$ ($\underline{d}^+$) that appears in any downward path through the pomtree, and a^- is currently available to enable the $\underline{d}^+$ that appears in the immediately adjacent B branch.

We say that two partial executions α, $\beta \in \pi(P)$ are execution equivalent, written α

$\equiv_e \beta$, when their sets of possible futures are equal, that is, $f_P(\alpha) = f_P(\beta)$ [5]. This is standard pomset (or pomtree) equality, based on lpo isomorphism. Each $\equiv_e$ equivalence class of partial executions is an execution state of process P. In the same way, each $\equiv_b$ equivalence class ($\alpha \equiv_b \beta$ when their sets of possible socket-extended futures are equal) is a behavior state of process P. Formally, if α and β are prefixes of infinite paths through pomtree P, then α and β result in the same behavior state of P precisely when the two pomtrees P/α and P/β descending from α and β are equal, where P/α and P/β have been extended to include the $(\alpha, P/\alpha)$ and $(\beta, P/\beta)$ precedences, and isomorphism now requires matching both event labels and arrow labels. In spite of the name, behavior states are not states.

3. Correct implementation

An implementation may exceed the minimum requirements of the specification. For example, it may be more liberal in accepting input and more conservative in producing output, but only if all progress requirements are satisfied. In model checking, a straightforward way to define correctness is to use the mirror of the specification as a conceptual implementation tester [1]. That is, one forms an imaginary closed system by linking mirror mP of specification P to the implementation -- in the general case, a network of processes *Net* --, and then examining certain properties of the resulting system. In the partial order world, the mirror is formed merely by inverting the causal/noncausal interpretation of P's successor arrows. Many things may now be examined. Is there a failure somewhere, causing the system to become undefined? Does the system just stop, violating fundamental liveness? Is some progress requirement violated? Is some conflict resolved unfairly? To study the closed system S = mP|P′ -- in the general case, S = mP|*Net* -- by partial order methods, we first regroup all successor arrows (as shown below) to obtain the system causal and noncausal successor relations. Graph predicates are then evaluated on a system pomtree that has two distinct successor relations.

In closed system S, each action is attributed to precisely two processes as a joint action of those two processes. System behaviors are projected onto component alphabets to yield component behaviors. Consider first the trivial case that the implementation is a process. Conceptual execution of S = mP|P′ produces corresponding partial executions of specification P and implementation P′. These partial executions correspond by containing the same events, but may not agree on which temporal precedences are necessary. For $\alpha \in \pi(P)$ and $\alpha' \in \pi(P')$, $\alpha \Longleftrightarrow \alpha'$ denotes correspondence. Each α,α' pair is a "doubly-ordered" finite pomset $[V, \Sigma, \Gamma, \Gamma', \mu]$. Corresponding pairs are produced by conceptual execution of S only as long as no safety or liveness violation is detected.

Safety checking occurs on two levels. (i) Any action u that is causally but not noncausally enabled at α,α' is an immediate safety violation. Sets of concurrently causally enabled actions must be concurrently noncausally enabled. (ii) Moreover, while visiting event u, each noncausal arrow (t, u) is checked by a (breadth-first) search for a supporting chain [t, u] of causal arrows. The nonexistence of such a chain is a safety violation. Liveness checking also occurs on two levels. (i) Any external output action v that is noncausally but not causally enabled at α,α' is an immediate liveness violation -- unless v is only disjunctively required (see above). Each set of concurrently causally enabled external output actions must contain some set of concurrently noncausally enabled external output actions. (ii) Moreover, while visiting external output event v, each causal chain [t, v] from an external input event t is checked by a "search" for a matching noncausal arrow (t, v). The nonexistence of such an arrow is a liveness violation.

We give a state-based definition of correctness for S = mP|P′. First, consider determinate P and P′. Correctness of safety properties means: $\forall\alpha,\alpha' : \alpha \Longleftrightarrow \alpha' : in_p(\alpha) \subseteq in_{p'}(\alpha')$ and $out_p(\alpha) \supseteq out_{p'}(\alpha')$. Correctness of progress properties means: $\forall\alpha,\alpha' : \alpha \Longleftrightarrow \alpha' : req_p(\alpha) \subseteq req_{p'}(\alpha')$. In restricted process theory, this reduces to: $out_p(\alpha) =$

$out_{p'}(\alpha') = req_p(\alpha) = req_{p'}(\alpha')$. Next, extend consideration to nondeterminate P and P'. In restricted process theory, correctness now means:

$$\forall\alpha,\alpha' \quad : \alpha \Longleftrightarrow \alpha':$$
$$\forall j \quad \exists j' \quad : \ in_j(\alpha) \subseteq in_{j'}(\alpha') \ \wedge \qquad (*)$$
$$\forall k' \quad \exists k \quad : \ out_k(\alpha) = out_{k'}(\alpha') \qquad (**)$$

These subscripts index sets of actions that are concurrently enabled at α, respectively, in each of the futures that branch from α.

We are now ready for a genuine partial order view that defines correctness without reference to states; in particular, the $\forall\alpha,\alpha'$ quantifier will be dispensed with. Consider a pair α,α', $\alpha \Longleftrightarrow \alpha'$. We regroup the causal and noncausal parts of $\Omega(\alpha)$ and $\Omega'(\alpha')$ to obtain the system causal and noncausal successor relations. We define the system causal successor relation $\tilde{\Omega} = (N' \cup \Xi)^{+-}$, where the superscript denotes transitive closure followed by transitive reduction. Similarly, we define the system noncausal successor relation $\hat{\Omega} = (N \cup \Xi)^{+-}$. When the implementation is a network *Net* of components P_i, $i \in I$, $\tilde{\Omega} = [(\cup_i N_i) \cup \Xi]^{+-}$, and $\hat{\Omega} = [N \cup (\cup_i \Xi_i)]^{+-}$.

Consider the general case. System pomtree S is undefined if a component fails. Suppose there is no failure (safety violation) in S. Each event v in S is classified as an input event in precisely one (say P_i) of the two processes to which the action performed by event v is attributed. From P_i's specification, there is a well-defined noncausal preset in P_i (say $\Pi_i(v) \subseteq \Sigma_i$) of the P_i action performed by event v. By definition, each event occurs in S because it is causally enabled. In order that P_i not "explode", v must also be noncausally enabled in P_i. This is equivalent to: (i) $\forall u \in \Pi_i(v) : u \in S$, and (ii) for each u, there is a chain of causal arrows [u, v]. This is safety correctness. Some events v that are noncausally enabled in S are, from specification P's point of view, classified as (external) output events. From P's specification, there is a well-defined noncausal preset in mP (say $m\Pi(v) \subseteq m\Sigma$) of the mP action performed by event v. In order that (the nonoccurrence of) event v not be a progress violation, we must have: (i) $v \in S$, and (ii) there exists a chain of causal arrows [u, v] iff $u \in m\Pi(v)$. Condition (ii) means that, after projection of S on mP, precisely $m\Pi(v)$ enables the mP action performed by v. Again, v may be only disjunctively required (see above). This is progress correctness.

4. Model checking

We sketch direct model checking of networks of processes. Given are specification process P and some number of implementation component processes P_i, $i \in I$. Network links map each internal output action of each component process P_i to precisely one internal input action of some other component process P_j [6]. Each internal event is attributed to two component processes. Conceptual links map each action of specification process P, whether input or output, to precisely one action (of the same type) of some component process P_i. Each external event is attributed to the specification process and one component process. External input events are caused by specification process P, while all other events are caused by some component process P_i.

The verification procedure shares superficial structure with the algorithm in [1]. Let D be a complete set of loop cutpoints (dominator set) for P. We enumerate system pomtree S recursively. We maintain (1) a stack to postpone the examination of system pomtree branches (maximal determinate behavior segments), and (2) a table to detect cycles in the enumeration of S. Recording the system behavior state each time P

advances to some d ∈ D (at P command completion) leads to discovery of a loop cutpoint for each loop in a finite presentation of system pomtree S.

Fig. 4 shows a typical result of applying a P command to a network, here, in its initial state.

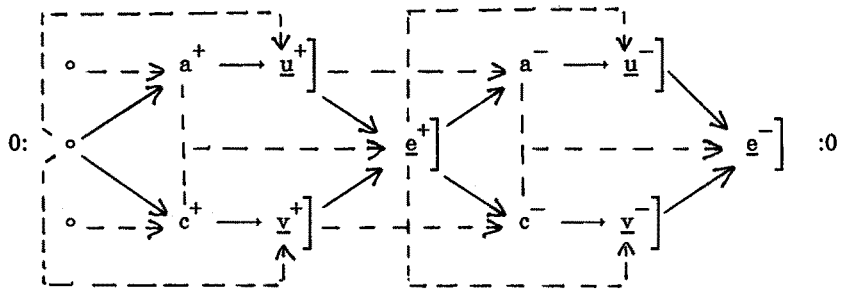

Fig. 4 Direct verification of a network.

Conceptual execution of system S produces n-tuples $<\alpha, \alpha_1, \alpha_2, \ldots>$ obtained by projection of system behavior prefixes onto processes P and P_i, $i \in I$. In the general case, corresponding partial executions no longer contain the same events. Each $\beta \in \pi(S)$ maps onto a system behavior state, which is the standard vector of behavior states of P and all P_i. Distinguish a "red" problem and a "green" problem. The red problem is, identify each maximal determinate behavior segment of system pomtree S. Branch points of S are scheduled for expansion in the usual way. The green problem is, identify each maximal behavior prefix $\beta \in \pi(S)$ that corresponds to a selected behavior state of specification P. System behavior states of such β's are candidates for entry into the table. The solution to both problems is, conceptually mark the following events in the indicated manner: (1) each initial event of each P command emanating from an input (output) branch point is marked with both a red and a green dot (with a green dot only), (2) each initial event of each P_i command emanating from an output branch point is marked with a red dot only, and (3) each initial event of each P command emanating from a determinate loop cutpoint is marked with a green dot only.

We move forward cleanly to a branch point of S by deferring visiting any red event as long as there are nonred events still enabled. Similarly, we move forward cleanly to a system behavior prefix $\beta \in \pi(S)$ that corresponds to a P rule completion by deferring visiting any green event as long as there are nongreen events still enabled. If the (selected) P behavior state of β has been identified as a loop cutpoint in P, then the system behavior state of β is entered into the table. If this state is already in the table, then β is not extended further.

Let $r : \pi(S) \rightarrow \pi(P)$ be the standard projection from system behavior prefixes to specification behavior prefixes. Refinement mapping is nontrivial because r has too many inverse images. The function of green dots is to select only a few "extremal" inverse images of selected behavior states of P -- to create (potential) table entries. For example, some P commands terminate in output-branch loop cutpoints. This is the case for command 1 in the full arbiter behavior machine, with a typical implementation being a ring of DME elements. Command 1 is applied to this implementation by moving downward along all paths of system pomtree S to include any event -- that is causally enabled by a^+ and b^+ -- up to but not including any green event (here, $\underline{c}^+$ and $\underline{d}^+$). With this implementation, command 1 of P corresponds to a finite initial pomtree prefix of S rather than a determinate system behavior segment. In our table, we record the two system behavior states that correspond to the endpoints of the two maximal paths through this finite system subpomtree.

Here is the algorithm summary. Recursively generate the system pomtree S by extending all system behavior prefixes $\beta \in \pi(S)$ as long as no safety or liveness violation is detected. Starting from the initial global state, apply P commands to the implementation by generating all causally enabled events in all possible futures up to but not including any green event. As events are generated, evaluate the graph predicates discussed in section 3. At P command completion, if the postlabel t in s: c :t belongs to the dominator set D of P, then record all pairs (t, N), where N is any network behavior state that corresponds to t by the construction in the previous paragraph. Do not apply a command with prelabel t to the implementation in network behavior state N if (t, N) is already in the table. Terminate the algorithm when it is no longer possible to apply any P command.

5. Conclusion

Behavior machines -- finite presentations of pomtrees -- precisely describe the branching and recurrence structure of processes, with benefits for both analysis and synthesis. Partial-order model checking allows us -- whenever there is a moderate amount of true concurrency but not an excessive amount of true nondeterminism -- to avoid the state explosion that frequently haunts verifiers based on state graphs, and limit any combinatorial explosion to that created by the branching structure of processes. Loop cutpoint detection in behaviors appears to require behavior states; such states contain information about how future behavior follows past behavior. Based on an understanding of the state space, the specifier/designer can create a useful separation of concerns for the verifier by structuring processes into (i) determinate behavior segments that are either completed or eliminated during conceptual execution (a pretending atomicity property), and (ii) input or output branch points. We obtain a dramatic performance improvement in model checking, and this even though only the specification process P gets to pretend atomicity.

References

[1] D.L. Dill, "Trace theory for automatic hierarchical verification of speed-independent circuits", Ph. D. Thesis, Department of Computer Science, Carnegie Mellon University, Report CMU-CS-88-119, February 1988. Also MIT Press, 1989.

[2] Z. Manna and A. Pnueli, "Specification and verification of concurrent programs by ∀-automata", Proc. of 14th ACM Symposium on Principles of Programming Languages, January 1987, pp. 1-12.

[3] A.J. Martin, "Compiling communicating processes into delay-insensitive VLSI circuits", Distributed Computing, Vol. 1, No. 4, October 1986, pp. 226-234.

[4] V.R. Pratt, "Modelling concurrency with partial orders", Int. J. of Parallel Prog., Vol. 15, No. 1, February 1986, pp. 33-71.

[5] D.K. Probst and H.F. Li, "Abstract specification of synchronous data types for VLSI and proving the correctness of systolic network implementations", IEEE Trans. on Computers, Vol. C-37, No. 6, June 1988, pp. 710-720.

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

[7] D.K. Probst and H.F. Li, "Partial-order model checking of delay-insensitive systems". In R. Hobson et al. (Eds.), Canadian Conference on VLSI 1989, Proceedings, Vancouver, BC, October 1989, pp. 73-80.

[8] J.v.d. Snepscheut, "Trace theory and VLSI design", Lect. Notes in Comput. Sci. 200, Springer Verlag, 1985.