PARTIAL-ORDER MODEL CHECKING: A GUIDE FOR THE PERPLEXED

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

ABSTRACT

Practicing verifiers of finite-state concurrent systems should be able to adapt our partial-order methods for verifying delay-insensitive systems to other verification problems. We answer the question, is it possible to control state explosion arising from various sources during automatic verification (model checking) of delay-insensitive systems? State explosion due to concurrency is handled by introducing a partial-order representation for processes, and defining system correctness as a simple relation between two partial orders on the same set of system events. State explosion due to nondeterminism is handled when the system to be verified has a compact, finite recurrence structure. Backwards branching through representations is a further optimization. In system verification, we start with models of system components that explicitly distinguish concurrency, choice and recurrence structure; during model checking, this a priori structure of components allows us to construct a compact, finite representation of the specification-constrained implementation -- without prior composition of system components. The fully-implemented POM verification system has polynomial space and time performance on traditional asynchronous-circuit benchmarks that are exponential in space and time for other verification systems; in general, the cost of running our verification algorithm is proportional to the size of the constructed system representation.

Keywords delay-insensitive system, model checking, state explosion, partial-order representation, recurrence structure, state encoding.

1. Introduction

Delay-insensitive systems are motivated *inter alia* by difficulties with clock distribution and component composition in clocked systems [1,2,6,11]. In a delay-insensitive system, modules may be interconnected to form systems in such a way that system correctness does not depend on delays in either modules or interconnection media. An important question is, what restrictions must be placed on finite-state concurrent systems in order that efficient model-checking algorithms are possible? Among concurrent systems, delay-insensitive systems have the particularity of containing regular patterns of handshakes between input and output events. Our work on delay-insensitive model checking can be adapted to handle delay-constrained reactive systems where inputs enable outputs, and outputs enable inputs [10]. Delay insensitivity has a natural link to controlling state explosion during automatic verification; the simple enabling relations in delay-insensitive systems make it easy to discover a solution to the state-explosion problem based on causality checking. To build an efficient automatic verifier based on causality checking, you need the following items for each system component: (i) an expressive finite partial-order representation that explicitly distinguishes concurrency, choice and recurrence, and (ii) a "goal-directed" state encoding that is both causality comprehensive (includes all causality) and state minimal (has fewest states). Given this, you can combine the best features of automata- and partial-order-based methods, and obtain a verification algorithm whose cost in space and time is proportional to the size of the constructed system representation. This size is often polynomial in the number of system components.

The automata we use to represent processes are called behavior automata, which can be unrolled to produce infinite event structures called pomtrees. The latter are essentially sets of partially-ordered computations where the branching structure due to conflict resolution has been made explicit [6-10]. Partial orders and schedule/automaton duality are covered in [3-4]. Restrictions on behavior automata trade off between expressiveness and processability (e.g., the efficiency of verification algorithms).

This research was supported by the Natural Sciences and Engineering Research Council of Canada under grants A3363, A0921 and MEF0040121. Email: probst@crim.ca.

The following method allows us to keep the termination table small. We provide a behavior automaton that distinguishes concurrency, choice and recurrence structure for each implementation component and for specification P. We label arrows in behavior automata to encode the process state corresponding to an arbitrary partial execution. System correctness is defined as a simple relation on an "enriched" system pomtree S that contains both causal and noncausal partial orders. This pomtree represents the imaginary closed system (also called S) produced by linking the mirror mP of specification P to the implementation network *Net*. During model checking, we use the a priori information about component structure to define a small set of loop cutpoints in a finite representation of system pomtree S, implicitly constructing a behavior automaton for the specification-constrained implementation. The explicit structure allows a convention to be followed during model checking that makes the mapping from P states to S states one-to-few rather than one-to-many, leading to a small termination table. Intuitively, when we cycle in P, we can arrange to cycle in S. Results obtained since [8] include: (i) simpler views of correctness and state, (ii) backwards branching in behavior automata, and (iii) a clear statement of algorithm complexity.

2. Abstract specification of asynchronous processes

Our specifications define externally-visible computational behaviors of processes; in partial-order representations, they specify precedence constraints [5,6]. A process P has a set of input ports and a set of output ports; a process action is a (port, token) pair, where the token represents a control or data value. Each performance of an action is a separate event. Processes are modelled by pomtrees, which are identical to computation trees except that their arcs are finite posets, and their vertices are input or output choice points. A process behavior $p \in P$ is a maximal conflict-free set of events of P, i.e., some full use of P by P's environment. If P is a process, then P's input actions are under the control of P's environment, while P's output actions are under the control of P. A requirements specification of a reactive process with asymmetry of control for input and output need not be equivalent to an ω-regular language containment problem of the traditional kind.

Safety properties (in DI systems, precedence properties) constrain both the process and its environment. A safety violation is the performance of an input or output action that is not enabled. A process receiving unsafe input logically fails ("explodes"). Liveness properties (in DI systems, response properties) constrain only the process. A liveness (progress) violation is the nonperformance of a required process output action. Fairness properties also constrain only the process; they assert fairness of conflict resolution in repeated process choice among output alternatives. Such fairness is the default assumption in all our specifications. Behavior automata allow integrated specification of safety and liveness properties, but fairness properties must be provided as a supplementary condition.

2.1. Pomsets

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 [3]. The successor relation Ω is the transitive reduction of Γ. A pomset is an isomorphism class of lpo's. Process behavior $p \in P$ is an infinite pomset. Process P is an infinite pomtree. Each behavior segment (pomtree arc) of process P is a finite poset. $\pi(p)$ is the set of finite prefixes of p. $p - \alpha$ is the suffix in p of $\alpha \in \pi(p)$. ${}^{o}p$ is the set of action labels of initial events of p. In a determinate (single-behavior) process, ${}^{o}(p - \alpha)$ is the set of actions concurrently performable after α. If α is a choice point in a nondeterminate (multiple-behavior) process, then there are sets of actions concurrently enabled after α, but not concurrently performable [6,9].

2.2. Behavior automata

Behavior automata are constructed in three phases. First, there is a deterministic finite-state machine (stick figure) D that expresses both conflict resolution (choice) and recurrence structure. D is a "small" automaton relative to the transition system dual to the pomtree [4]. Second, there is an expansion of dfsm transitions (sticks) into finite posets, with additional machinery (sockets) to specify nonsequential concatenation of posets. Third, there is an iterative process of labelling successor arrows in posets, which terminates with an appropriate state encoding.

We sketch the formal definition of behavior automaton. Given the disjoint alphabets Act (the set of process actions), Arr (the set of successor arrow labels), Com (the set of dfsm D transitions) and Soc (the set of sockets), first define Pos as the set of finite labelled posets over Act $\cup$ Soc. Each member of Pos is a labelled poset (B, Γ, ν), where (i) Γ is a partial order over B $\subseteq$ Act $\cup$ Soc, and (ii) ν: $\Omega \rightarrow$ Arr assigns a label to each element in the successor relation Ω (the transitive reduction of Γ). A <u>behavior automaton</u> is a 3-tuple (D, ϵ, ψ), where (i) D is a dfsm over Com, (ii) ϵ: Com $\rightarrow$ Pos maps dfsm transitions to labelled posets, and (iii) ψ: Soc $\rightarrow$ powerset(Act) maps sockets to sets of process actions. Map ψ defines which process actions can "plug in" to an empty socket when a poset command is concatenated to a sequence of earlier poset commands as defined by dfsm D. There is also an imaginary reset action $\bullet$.

A Petri net is uniquely characterized by the presets and postsets of its transitions. Similarly, sockets can be removed from a behavior automaton by concatenating commands, and considering predecessor and successor arrows of individual actions. A process action consumes its predecessor arrows (removes them from the old state), and produces its successor arrows (adds them to the new state). Behavior automata are typeset by writing the poset transitions separately, and using "digit colons" to identify dfsm vertices. These vertices are not state encodings. Commands are also called productions.

Fig. 1 shows a behavior automaton for a C-element. Each arrow in the production is assigned a distinct label. The semantics is straightforward. For example, action a^+ is enabled in any state containing arrow 1; when it is performed, arrow 1 is removed from the state and arrow 3 is added. Similarly, action $\underline{c}^+$ is enabled and required (because of the bracket) in any state containing arrows 3 and 4. When it is performed, arrows 3 and 4 are removed from the state and arrows 5 and 6 are added. Since $\psi(\circ)$ = {$\bullet$, $\underline{c}^-$}, action $\underline{c}^-$ has its preset and postset given by: {7, 8} $\underline{c}^-$ {1, 2}. The postset follows from the labels on arrows leaving $\circ$. For program convenience, behavior automata are decomposed into atoms of the form {predecessor arrows, action, successor arrows}, retaining the stick figure D as a structuring device.

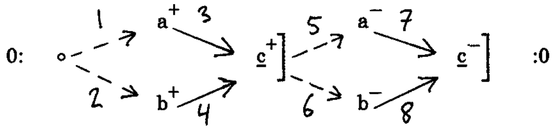

Fig. 1 Behavior automaton for a C-element.

The use of dashed and solid arrows is a reminder that a process specification includes both an <u>interprocess</u> protocol (given by the dashed arrows) and an <u>intraprocess</u> protocol (given by the solid arrows). Since the state is encoded as the set of arrows crossing a consistent cut, using fewer arrow labels would alter the enabling relation of the C-element; hence, this state encoding (arrow labelling) is fixed up to isomorphism.

Fig. 2 shows a behavior automaton for a delay-insensitive arbiter. Clients follow a four-cycle protocol. $\langle A \rangle = \underline{c}^+] \overset{2}{\dashrightarrow} a^-$ and $\langle B \rangle = \underline{d}^+] \overset{6}{\dashrightarrow} b^-$ are the two critical sections. The $\circ$ in command 1 is filled only by $\bullet$, i.e., $\psi(\circ)$ = {$\bullet$}. The top $\circ$ in command 2 is filled only by a^+, i.e., $\psi(\circ)$ = {a^+}. The bottom $\circ$ in command 3 is filled only by b^+, i.e., $\psi(\circ)$ = {b^+}. The middle $\circ$'s in commands 2 and 3 can be filled by $\bullet$, a^- or b^-, i.e., $\psi(\circ)$ = {$\bullet$, a^-, b^-}.

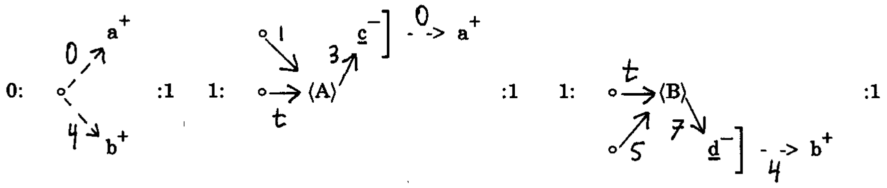

Fig. 2 Behavior automaton for a delay-insensitive arbiter.

When sockets have been removed, this set of actions generates the transition system of the fifteen-state arbiter dfsm [0]. However, care must be taken to keep the state encoding causality comprehensive. Arrow 1 in the C-element is two arrows in disguise ($\bullet$ - -> a^+, $\underline{c}^-$ - -> a^+); this is not a problem because of the special nature of $\bullet$. Arrows 0 and 4 in the arbiter are similar, but arrow t is <u>six</u> arrows in disguise (three sources, two sinks). Consider performing action $\underline{c}^+$ in state {1, 5, t}. Checking arrow t requires backing up in the behavior automaton to both process-action sources of t, viz., a^- and b^-. Again, $\bullet$ is not a problem. These are distinct causality arrows that must be checked separately. An arrow with multiple process-action sources is called an <u>equivalenced</u> arrow; all its sources are recorded in a table. In state {1, 5, t}, $\underline{c}^+$ and $\underline{d}^+$ are concurrently enabled but conflicting actions; hence, causality checking in this state requires both forwards and backwards branching through the behavior automaton.

2.3. <u>Restrictions on processes</u>

Which finite-state concurrent systems can be model checked efficiently? The efficiency gains in our algorithms stem from two sources: (i) making causality checking the primary activity, and (ii) recording as few system states as possible. Delay insensitivity <u>per se</u> is not a precondition for causality checking. However, well-behavedness conditions do restrict the class of event structures on which our programs operate. Among these are rules that allow processes to be used as components of delay-insensitive systems [6,11]. Each rule is a genuine restriction on concurrent systems. It is an important <u>metarule</u> that branching and recurrence structure be made explicit.

<u>General rules</u>

<u>Rule a1</u> There is no autoconcurrency. Formally, any two events at the same port in $p \in P$ are ordered by Γ. (A good general rule).

<u>Rule a2</u> Processes are finite state. Formally, there is an upper bound on the number of Ω arrows crossing a consistent cut of any $p \in P$. (Arrow labels can then be used to encode the process state).

<u>Rule a3</u> A process has a representation with a compact, finite recurrence structure as defined by its dfsm D. Given finiteness, it is enough to require the independence of concurrent choices. Formally, if two choices are concurrent (causally independent), then neither can affect the existence or outcome of the other. (A reasonable general assumption).

<u>Delay-insensitivity rules</u>

<u>Rule b1</u> There is handshaking between any two successive events at the same port. 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. (A DI-specific rule related to zero buffering).

<u>Rule b2</u> There is no specified successor relationship between two input events or two output events. Formally, each line in $p \in P$ consists of an infinite sequence of strictly alternating input and output events. (Simplifies causality checking by introducing a simple notion of preset).

<u>Rule b3</u> If a set of enabled output actions can be performed concurrently, then they, or some conflicting set of concurrently performable output actions, must be performed. Formally, if $p \in P$, then all output events in p are bracketed. (A reasonable assumption in hardware systems).

One might ask if requiring concurrent choices to be independent is similar to requiring that Petri nets be free choice; the answer is no (arbiters are not free choice). In partial-order representations, finding a reasonable restriction on the generality with which conflicts can occur is the key issue in trading off between decision power and modelling power. Behavior machines are "free choice" only at the level of dfsm D, which explicitly distinguishes conflict resolution and recurrence structure.

Our algorithms verify systems efficiently when all processes satisfy rules a1 through b3. Rule a1 eliminates artificial examples. Rule a2 is central to finite checkability. Rule a3 eliminates nonphysical nondeterminism. The simplest way to satisfy rule a3 is to start with a dfsm D composed exclusively of determinate behavior segments and input or output choice points. In principle, rules b1 and b2 could be replaced -- provided there is still some notion of cross enabling (input enables output, output enables input). Rule b3 promotes efficient checking.

2.4. Semantics of behaviors

Fig. 3 shows an unlabelled behavior automaton for a determinate process. Action labels are pure names, but output actions are underlined. The partial order in this behavior is the transitive closure of a successor relation $\Omega = N \cup \Xi$, where N is a relation from input events to output events, and Ξ is a relation from output events to input events. Because of control asymmetry, we say that N is a causal relation and Ξ is a noncausal relation. Each output (input) action has a causal (noncausal) preset defined by the sources of its incoming solid (dashed) arrows. A process may perform an output action when the causal preset has occurred. An environment may perform an input action when the noncausal preset has occurred.

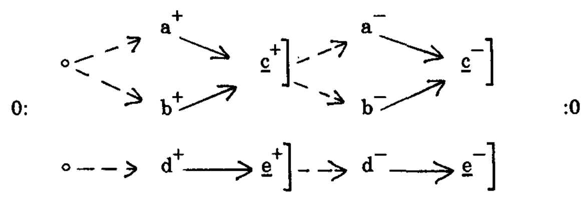

Fig. 3 Unlabelled behavior automaton (determinate process).

2.5. States in partial-order representations

The reduced finite-state machine for a delay-insensitive arbiter has fifteen states [0]; one of them is a choice state where the arbiter can give the token to either client. Reduction of the fsm means there is no record in the state of who returned the token. In the arbiter behavior automaton, either (i) we use different labels for different token arrows, or (ii) we use the same label (namely, t). Different labels distinguish the states resulting from the return of the token by different clients. Reduced fsms correspond to using equivalenced arrows. Verification by causality checking is complete when each distinct instance of causality has been checked. In particular, arrows leaving different critical sections must be checked separately. Nonreduced transition systems can be used to support completeness arguments for verification algorithms based on causality checking. This is the motivation for the distinction between execution state and behavior state [8]. In causality checking, behavior states can be used either explicitly (if equivalenced arrows are renamed to identify their sources) or implicitly (if tables of sources are kept). Conceptually, we explore all the distinct ways each process action can be enabled. Our current preference is to use equivalenced arrows and tables of sources; we perform "state-based" causality checking: an enabled action has its causality checked, not with respect to a particular past, but rather with respect to any past that would have resulted in the same state; in general, this requires backwards branching through systems of behavior automata.

3. Correctness as a graph predicate

We define correctness in a causality-based scheme by using the mirror mP of specification P as a conceptual implementation tester [1]. We form an imaginary closed system S by linking the mirror mP of specification P to the implementation network of processes *Net*. This produces an "enriched" pomtree of system events with two partial orders; system correctness is defined as a simple, easily-checked relation between the two orders. The intuitive notion of correctness is as follows, given that implementations may be input liberal and output conservative [6,9]. Is there a failure somewhere, causing system S to become undefined? Does the system just stop, violating fundamental liveness? Is some progress requirement of P violated? Is there (program-detectable) nondeterminate livelock in S so that an appeal to fairness of system components is necessary to assert progress? Is some conflict corresponding to output choice in P resolved unfairly?

Mirror mP is formed by inverting the type of P's actions and the causal/noncausal interpretation of P's successor arrows, turning P's dashed arrows into solid arrows and vice versa. Brackets are preserved unchanged. Every action that can be performed in S is a linked (output action, input action) pair. As a result, we can check whether intraprocess protocols support interprocess protocols in closed system S.

We bootstrap the dashed (noncausal, interprocess protocol) and solid (causal, intraprocess protocol) relations from process actions to system actions, defining an event structure called an "enriched" pomtree, with a noncausal enabling relation on top of the usual causal enabling one. For example, a noncausal predecessor of system action σ is found by locating the embedded process input action, stepping back along a dashed process arrow, and returning to the system alphabet. This defines the noncausal preset of a system action. Essentially, the safety correctness relation is: whenever a dashed arrow links two system actions, a chain of solid arrows must also link the two actions.

Let σ be a system action that is causally enabled in S. There is a safety violation at σ unless (i) its noncausal preset is causally enabled in S, and (ii) each member of its noncausal preset is a causal ancestor of σ. The causal preset of σ is defined only when σ is a bracketed system action: it is the set of nearest performances of linked mP output actions on any causal chain coming into σ. In order that a bracketed σ in S is neither a safety nor a progress violation, it is necessary that the causal and noncausal presets of σ match exactly. When backwards branching is used to resolve multiple sources of equivalenced arrows, conditions for either safety or safety/progress must hold in each distinct past (backwards branch).

Fig. 4 is a simple illustration of correctness with only external events; solid arrows cover up dashed arrows in the system figure (covered dashed arrows are trivially supported). The dashed arrow from a to c in S is supported by a chain of solid arrows, so there is no safety violation at c. However, the noncausal preset of c, viz., {a}, does not match the causal preset, viz., {d}, so there is a progress violation at c.

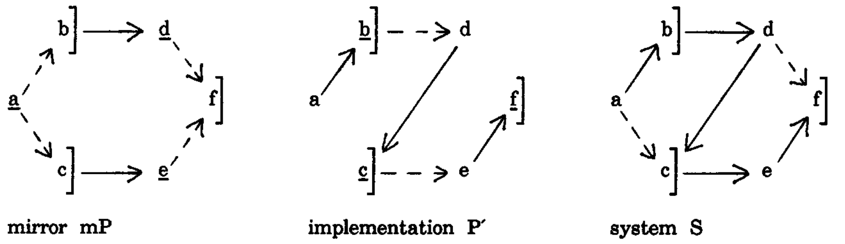

Fig. 4 Partial orders and total correctness of an open system as a graph predicate.

Fig. 5 shows a second illustration. Every dashed arrow is supported by a chain of solid arrows, and all causal and noncausal presets of bracketed system actions match exactly. The implementation is correct. Although safety correctness is containment of the causal language in the noncausal language, this is not implementation language in the specification language. Since S represents the specification-constrained implementation, coupled system processes can prune each other's branching structure; this is the problem with traditional language containment.

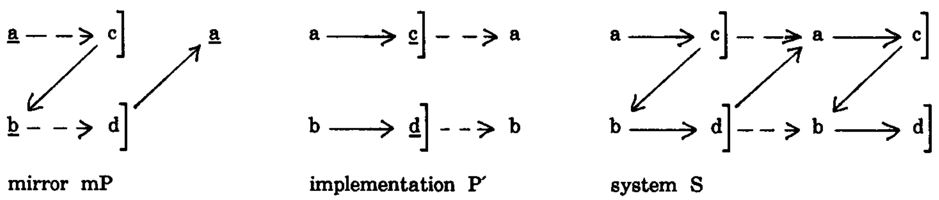

Fig. 5 Partial orders and total correctness of an open system as a graph predicate, bis.

4. Model checking

The algorithm is also straightforward. We enumerate system actions and visit one system cut per action. We consider each enabled action in the context of a state we have reached. First, we repeatedly step back across single dashed arrows to compute the action's noncausal preset. Second, we repeatedly (finitely) chain back across multiple solid arrows to compute the action's partial causal ancestor set, or causal preset if the action is bracketed. When equivalenced arrows are encountered, we branch backwards to check each possible source. The speedup is due to two factors: (i) we effectively check cuts (through the generated past) we have not visited, and (ii) for equivalenced arrows, we effectively check cuts in pasts we have not generated. This kills state explosion due to concurrency and nondeterminism. We traverse each determinate segment (stick) of the implicitly constructed system behavior automaton (stick figure) precisely once. Backwards branching catches all causality that would have arisen had we traversed the system stick figure in some other way.

We keep the termination table small by making the mapping from P states to S states one-to-few rather than one-to-many. This is possible when all behavior automata have visible branching and recurrence structure. Hence, when we cycle in P, we can arrange to cycle in S. The top level of the algorithm visits system actions and tries to complete P sticks. The lower level of the algorithm does arrow checking.

The following algorithm assumes there is additional machinery to detect determinate livelock, or that such detection is not a problem. Logically, checkpoints are taken at system states corresponding to specification loop cutpoints. But which system states correspond to specification states, given that any number of internal system actions can be performed without performing any external system action? A branching loop cutpoint in P corresponds to some number of causal branch points in S. A nonbranching loop cutpoint in P can be made to correspond to one nonbranching loop cutpoint in S. For each loop cutpoint in P, we identify a small number of system states we can consistently come back to. For this purpose, checkpoints are taken at (i) unique system states corresponding to nonbranching loop cutpoints in P, and (ii) causal branch points in S. We make each specification loop cutpoint, whether determinate (green problem) or nondeterminate (red problem), correspond to as few system states as possible, to minimize the number of entries in the termination table. Execution continues until no new checkpoints are found.

During model checking, we identify maximal determinate behavior segments of system pomtree S (the red problem). We do this by recognizing causal choice actions σ in closed system S. Each initial action of a P_i production leaving an output branch point, and each initial action of a P production leaving an input branch point, is marked with a red dot. Each causal choice action σ in system S is marked with a red dot. To ensure termination, we must also identify maximal path prefixes $\beta \in \pi(S)$ that correspond to determinate loop cutpoints in P's behavior automaton (the green problem). When we return to a P dfsm vertex, we want to return to as few system states as possible. Each initial action of a P production leaving a determinate loop cutpoint is marked with a green dot. Since P productions are the highest level of control for the performance of S actions, each initial system action σ of an S production leaving a determinate or nondeterminate loop cutpoint in S's (constructed) behavior automaton is now marked with a red or green dot.

We move forward cleanly to a branch point of S by not performing any red action as long as there are nonred actions still enabled. Branch points of S are scheduled for expansion in the usual way. Similarly, we move forward cleanly to a system path prefix $\beta \in \pi(S)$ that corresponds to a determinate loop cutpoint in P by not performing any green action as long as there are nongreen actions still enabled. Whenever (constructed) system determinate loop cutpoints or system causal branch points are encountered, the system state -- the vector of (mP, P_1, ..., P_n) states -- is entered into the termination table, using the chosen process-state encoding. If the system state is already in the table, then system path prefix β is not extended further.

During verification, we perform actions that are causally enabled in the closed system, starting from initialization. When we perform a system action, we update the state of both processes to which the action is attributed. We perform these actions in arbitrary order, subject only to the constraint that unmarked actions are performed before marked actions. As we perform actions, we check the graph predicate of the previous section. Occasionally, we discover a safety violation immediately, i.e., without any graph searching: a system action is causally enabled, but the corresponding input action is noncausally not enabled, i.e., forbidden. This is the only discovery mode in model checking based on sequence enumeration. More typically, we compute the noncausal preset of the system action (from the behavior automaton of the process that owns the input action) and search backwards in the system computation to determine whether each member of the noncausal preset is a causal ancestor of the system action.

When a bracketed system action is causally enabled (and is not an immediate safety violation), it is more efficient to do combined safety/liveness checking. The noncausal preset is computed as in the previous paragraph. From the semantics of brackets, an mP dashed arrow can only be supported by a chain of solid arrows not containing an mP solid arrow. Using this, we search backwards in the system computation along each causal chain to find the first performance of an mP output action, thereby computing the causal preset. In order that there not be a safety/liveness violation, the two presets must be equal. Exploration of a system segment that begins with marked actions may cause a new specification production to be loaded. Failure to complete this production, once loaded, indicates the presence of progress violations not detectable by combined safety/liveness arrow checking; this special kind of progress violation occurs when actions performed in S form a noncausal preset of an mP input action, but do not causally enable the linked output action. The moving algorithm completes system sticks (modulo violations of fundamental liveness), and tries to complete P sticks, once they have been loaded. Two ideas make this algorithm work: (i) maintain the asymmetry of mP within S; use the special semantics of bracketed system actions, and use P productions as templates to schedule the performance of system actions, and (ii) construct a refinement mapping from P states of P dfsm vertices to system states in which the image sets are as small as possible. State-based causality checking (backwards branching in behavior automata) is a further optimization.

5. Empirical results

Benchmarks are a double-edged sword; your program may work well only when it happens to work well. To some extent, this state of affairs can be remedied by a theory of the causes of state explosion that defines the class of concurrent systems that can be verified efficiently. A more concrete result is that the cost in space and time of our verification algorithm is proportional to the size of the constructed system representation; when this is polynomial, our verification algorithm is polynomial. Our two primary benchmarks (ring of DME's, n-place buffer) were chosen because both producer/consumer and mutual-exclusion solutions are fundamental building blocks of concurrent systems. Since the polynomial cost functions are clearly benchmark sensitive, there is a clear need for a suite of benchmarks.

A complete verification package has been written by Lin Jensen in the Trilogy programming language running on an IBM PC. The POM system has polynomial space and time performance on benchmarks that are exponential in space and time for other verification systems. Consider the ring of DME elements benchmark. The runtime for verification of both safety and liveness properties is quadratic in n, the number of DME elements. The number of system states grows exponentially with n. For example, when $n = 9$, the time is 180 s (roughly 10^9 states); when $n = 10$, the time is 220 s (roughly 10^{10} states). The space requirements for these problems do not exceed 64K bytes, i.e., one IBM PC data segment. What are the asymptotic space requirements? One must store the input; this is linear. One must store the termination table; this is quadratic. Given reasonable garbage collection, the working storage to do backwards chaining in a system computation is linear, because we construct and compare simple presets. The asymptotically limiting resource is the quadratic space for the termination table. With 64K bytes, we never enter the asymptotic region; the linear space term predominates. Fig. 6 illustrates the algorithm's complexity on this benchmark more graphically. The size of the termination table is the number of vertices (•'s) times the size of a system state, which is $\Theta(n)$; this gives $\Theta(n^2)$ for space. The runtime is the time taken to traverse each stick of the system stick figure precisely once. The number of sticks grows linearly, and the time to traverse a variable-size stick is $\Theta(n)$; this gives $\Theta(n^2)$ for time.

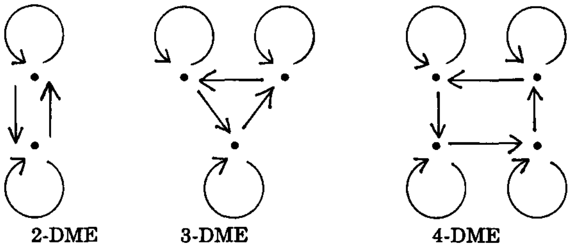

Fig. 6 System stick figures for the n-DME verification problem.

6. Conclusion

For a given benchmark, the state encoding strategy is the primary determinant of the number of system sticks that must be examined. Heap exhaustion (within the compiler-imposed bound of 64K bytes) obviously limits the size of problems we can verify. To facilitate distribution and further experimentation, we will rewrite the POM system in a more widely-used programming language such as LISP or C. It is easy to explain why rings of DME's cause such problems for traditional sequence-based verification algorithms, but not for ours; given a system stick figure, it is clear that it can be traversed efficiently, and that successful termination implies total correctness of the implementation. The nontrivial part of our approach is the predefined strategy for constructing a set of system vertices, which are not given beforehand. This strategy is explained in section 4, and may be essential in any efficient partial-order algorithm for network verification. If branching and recurrence structure are not provided for components, then we lose the one-to-few mapping from P states to S states. We would then need a replacement strategy for constructing a "small" automaton for S (this allows recording few system states). Other partial-order approaches to model checking, based in part on pruning pomset languages prior to checking language containment, appear to be less powerful in controlling state explosion. In any case, they address the problem of checking safety and liveness properties of (presumably large) precomposed closed systems, while we check the safety and liveness properties of (presumably large) open networks of processes.

Space is generally considered to be the critical resource in automatic verification. The asymptotic space complexity depends on the number of vertices in the constructed system behavior automaton. The asymptotic time complexity depends on the number and search complexity of individual sticks. Are there system behavior automata whose size is exponential in the number of system components? A simple-minded application of our techniques to show that there is no deadlock in a system of n dining philosophers with a circulating poison pill (or appetite suppressant) would examine a choice state in which $n - 2$ forks choose, producing as many branches as there are subsets of an $(n - 2)$-set. A priori, none of them could result in deadlock, so there is no point in generating exponentially-many branches. The space complexity is not a problem because there is only one choice state. The time complexity could conceivably be a problem if one were forced to explore every branch. A suite of nontrivial benchmarks for dining philosophers problems (something less symmetrical than deadlock detection) might allow meaningful comparison of different partial-order approaches. Combinatorial explosion of system behavior automata is a fruitful topic for future study.

References

[0] D.L. Black, "On the existence of delay-insensitive fair arbiters", Distributed Computing, Vol. 1, No. 4, October 1986, pp. 205-225.

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

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

[4] V.R. Pratt, "Modelling concurrency with geometry", Proc. 18th Ann. ACM Symposium on Principles of Programming Languages, January 1991, pp. 311-322.

[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] 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, June 1990, DIMACS Series, Vol. 3, 1991, pp. 15-24. Also Lect. Notes in Comput. Sci., Springer Verlag, forthcoming.

[9] D.K. Probst and H.F. Li, "Modelling reactive processes using partial orders". In M. Kwiatkowska et al. (Eds.), Semantics for Concurrency, Leicester 1990, Leicester, UK, July 1990, Workshops in Computing, Springer Verlag, 1990, pp. 324-343.

[10] 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". In S. Whitaker, (Ed.), Third NASA Symposium on VLSI Design, Moscow, ID, October 1991, Proceedings, pp. 8.2.1-8.2.8.

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

[12] J.T. Udding, "A formal model for defining and classifying delay-insensitive circuits", Distributed Computing, Vol. 1, No. 4, October 1986, pp. 197-204.

TABLE I

Time to verify an n-arbiter
implemented by a ring of n DME's
(n system states in the table)

n	seconds
2	20.60
3	32.08
4	47.13
5	65.63
6	87.77
7	114.03
8	144.51
9	179.60
10	220.03

TABLE II

Time to verify an n-buffer
implemented by n 1-buffers
(one system state in the 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