Controlling State Explosion During Automatic Verification of Delay-Insensitive and Delay-Constrained VLSI Systems Using the POM Verifier [1]

D. Probst and L. Jensen
Department of Computer Science
Concordia University
1455 de Maisonneuve Blvd. West
Montreal, Quebec Canada H3G 1M8

Abstract– **Delay-insensitive VLSI systems have a certain appeal on the ground due to difficulties with clocks; they are even more attractive in space. 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 systems, and defining system correctness as a simple relation between two partial orders on the same set of system events (a graph problem). State explosion due to nondeterminism (chiefly arbitration) is handled when the system to be verified has a clean, finite recurrence structure. Backwards branching is a further optimization. The heart of this approach is the ability, during model checking, to discover a compact finite presentation of the verified system 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. We also sketch the generalization of this approach to handle delay-constrained VLSI systems.**

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

1 Introduction

Delay-insensitive systems are motivated by difficulties with clock distribution and component composition in clocked systems [1,2,5,9]. 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. Gate-level implementations of modules whose specifications are delay-insensitive are often themselves quasi-delay-insensitive; essentially, the assumption of isochronic forks allows one gate to handshake on behalf of

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

another. Most interesting are delay-constrained reactive systems, in which either outputs or inputs or both must appear in some temporal window relative to enabling inputs or outputs. Hardware systems in space make delay insensitivity even more attractive due to *(i)* pervasive asynchronous communication, and *(ii)* extremely-low-power applications. Delay insensitivity has a natural link to controlling state explosion during automatic verification; the simple enabling relations in delay-insensitive control systems make it easy to discover a solution to the state-explosion problem based on causality checking. To build an automatic verifier based on causality checking, you need two things: *(i)* an expressive finite partial-order representation strategy that explicitly distinguishes concurrency, choice and recurrence, and *(ii)* a "goal-directed" state-encoding strategy that is both comprehensive (includes all causality) and minimal (has fewest states)—the last for performance reasons. Given these two things, you can combine the best features of automata-based and partial-order-based computational verification methods.

2 Behavior Automata

The basic automata used to represent processes are called behavior automata, which can be unrolled to produce event structures (essentially sets of partially-ordered computations with all branching due to conflict resolution made explicit) [5-8]. Partial orders and concurrent computation are discussed in [3]. Restrictions on behavior automata trade off between expressiveness and processability (e.g., the efficiency of verification algorithms) [8]. The most important rules for delay insensitivity are (cf. [10]):

Rule 1 Any two events at the same port in a partially-ordered computation are order-separated by at least one event at some other port.

Rule 2 There is no immediate order relation between two input events or two output events. Each ordering chain is an infinite sequence of strictly alternating input and output events.

We seek abstract, i.e., black-box, specifications [4]. For this purpose, behavior automata are constructed in three phases. First, there is a deterministic finite-state machine (stick figure) that expresses both conflict resolution (choice) and recurrence structure. This is a "small" automaton relative to the full transition system. Second, there is an expansion of dfsm transitions (sticks) into finite posets, with additional machinery (sockets) to define possibly nonsequential concatenation of posets. Third, there is an iterative process of labeling successor arrows in posets, which terminates with an appropriate state encoding.

We sketch the formal definition of behavior automaton. Given disjoint alphabets Act (process actions), Arr (successor-arrow labels), Com (dfsm transitions) and Soc (sockets), first define Pos as the set of finite labeled posets over Act $\cup$ Soc. Each member of Pos is a labeled poset (B, Γ, ν), where *(i)* Γ is a partial order over B $\subseteq$ Act $\cup$ Soc, and *(ii)* ν: Ω $\rightarrow$ Arr assigns a label to each element in the successor relation Ω (the transitive reduction of Γ). A behavior automaton is a 3-tuple (D, η, o), where *(i)* D is a dfsm over Com, *(ii)*

η: Com → Pos maps dfsm transitions to labeled posets, and *(iii)* o: Soc → powerset(Act) maps sockets to sets of process actions. Map o 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.

A C-element has two input ports a and b, and an output port $\underline{c}$. Two actions are possible at a given port depending on whether the signal transition is rising (+) or falling (−). There is no conflict resolution (choice), and the recurrence structure of D is a simple loop. Transitions (sticks) concatenate sequentially in this example, shown in Fig. 1. Both the reset action and action $\underline{c}^-$ can fill the unique socket in this poset. Digit colons identify dfsm D vertices.

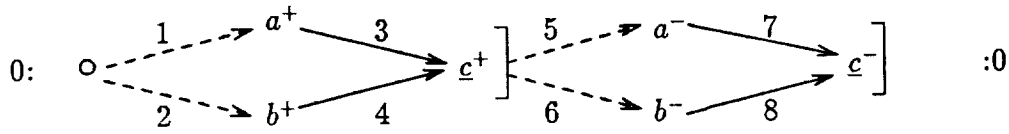

Figure 1: Behavior automaton for a C-element.

In the absence of conflict resolution, each enabled output action must be performed eventually (indicated by bracketing). The use of both dashed and solid arrows is a visual reminder that a process specification contains both an interprocess protocol (given by the dashed arrows) and an intraprocess protocol (given by the solid arrows). Here, the state encoding (arrow labeling) is essentially fixed; since the state is encoded as the set of successor arrows crossing from the past to the future, i.e., crossing a consistent cut produced by a partial execution, using fewer arrow labels would alter the enabling relation of the C-element.

The semantics are 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. Action $\underline{c}^-$ has preset and postset given by: {7, 8} $\underline{c}^-$ {1, 2}.

Behavior automata are more interesting when branching is involved. A delay-insensitive arbiter has two input ports a and b, and two output ports $\underline{c}$ and $\underline{d}$. It grants exclusive access to one of two competing clients at a time. The behavior automaton is shown in Fig. 2.

Clients follow a four-cycle protocol. $(A) = \underline{c}^+] \rightarrow a^-$ and $(B) = \underline{d}^+] \rightarrow b^-$ are the two critical sections. The labeling shown, if completed, would be conservative (the state encoding includes all causality, but is not minimal). Having arrows 8, 9 and 10 in state encodings indicates who made the token available (viz., first client, second client and reset action). These three arrows are distinct instances of causality that must be checked separately. Still, there are too many state encodings.

We can group arrows 8, 9 and 10 into an equivalence class t. This does not alter the enabling relation. Consider performing action $\underline{c}^+$ in state {1, 5, t}. Causality checking

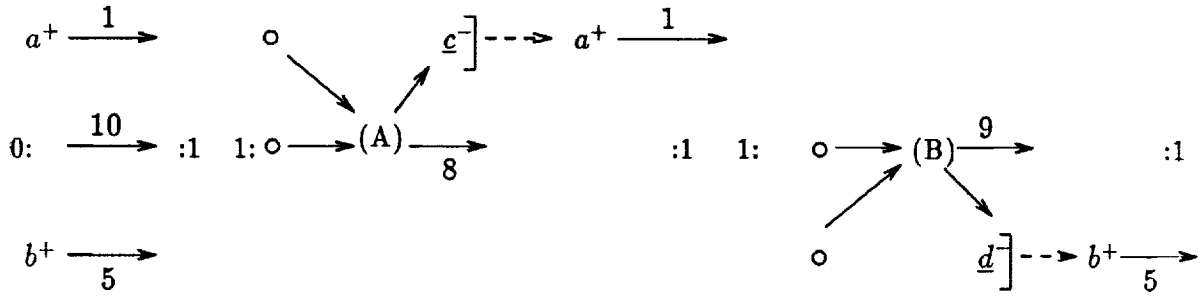

Figure 2: Behavior automaton for a delay-insensitive arbiter.

of arrow t requires backing up in the behavior automaton to both possible sources, viz., actions a^- and b^-. In state {1, 5, t}, $\underline{c}^+$ and $\underline{d}^+$ are concurrently enabled but conflicting actions. Verification algorithms that process behavior automata perform both forwards branching (conflict resolution) and backwards branching (examination of distinct recent pasts).

After equivalenced arrow t has been defined, we can complete the picture in Fig. 2 to make it match the formal definition (the labeled arrows leaving posets are derivable from map o). Consider the second poset command. The top socket is filled only by action a^+; its arrow is labeled 1. The middle socket is filled by any of the actions a^-, b^- and reset; its arrow is labeled t. The remaining (interior) poset arrows are given arbitrary distinct labels.

3 Correctness as a Graph Problem

We define correctness by using the mirror mP of specification P as a conceptual implementation tester [1]. We form an imaginary closed system S by linking mirror mP of specification P to the implementation network of processes *Net*. This produces an infinite pomtree (event structure) of system events on which two partial orders are defined; system correctness is then expressible as a simple, easily-checked relation between the partial orders. The standard model-independent notion of correctness is as follows. 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

(sometimes called 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. We have thus defined "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

(a) its noncausal preset is also causally enabled in S, and

(b) 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 present in S, these conditions are generalized to hold along each distinct past (backwards branch). Backwards branching is necessary to resolve multiple sources of equivalenced arrows.

4 Model Checking

The algorithm is straightforward. Starting from system reset, we enumerate causally-enabled system actions and visit one system cut per action. We consider each enabled action in a state produced by some partially-ordered past that we have generated. 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 effects:

1. we effectively check cuts in the generated past that we have passed by without visiting, and

2. for equivalenced arrows, we effectively check cuts in pasts that we have not generated.

This kills state explosion due to concurrency and/or 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 been visible had we traversed the system stick figure in some other way. Example system stick figures are shown in Fig. 3.

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

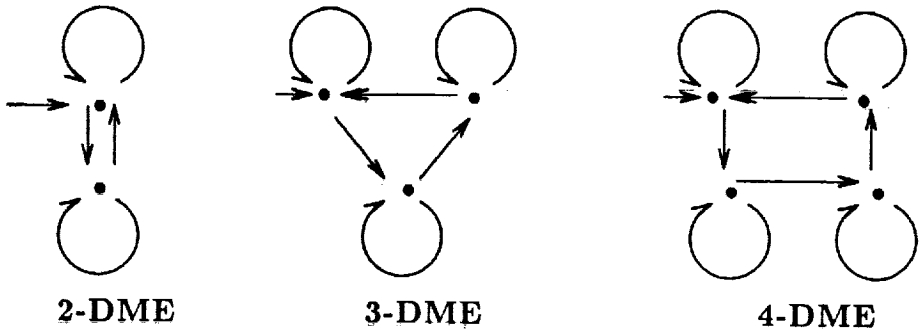

Figure 3: System stick figures for the n-DME verification problem.

visible branching and recurrence structure. Explicit structure in each component allows the verification algorithm to uncover a structure in system S. In particular, when we cycle in P, we can arrange to cycle in S. As a result, termination is achieved by checkpointing very few global states of system 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.

5 Output-delay-constrained reactive systems

To fix ideas, consider a hardware system that is a space-based component of a missile defense system; this component receives massive amounts of target-acquisition data asynchronously, and is required to process it in real time and communicate the result. There are two types of delay constraint that could appear in a requirements specification of such a component, which is a typical reactive system. First, there could be a temporal interval, relative to the arrival of a complete problem instance, during which the component must respond; this is an output delay constraint. Second, there could be a temporal interval, relative to the departure of the previous result and/or the arrival of other input, during which the external world can safely stimulate the component; this is an input delay constraint. The simplest delay-constrained reactive systems are those in which delay constraints are imposed only on the intraprocess protocol, i.e., on module response; in this case, the mechanism that ensures input safety is unchanged (the interprocess protocol is still real or virtual handshaking). The difficult case is an interprocess protocol that specifies when the module can be overwhelmed by high-bandwidth input; we leave the difficult case for future work. In our representation, minimum/maximum-delay information is expressed by putting timing windows directly on output actions. Minimum-delay information may be freely entered on successor arrows, but maximum-delay semantics is constrained by questions of physical realizability. We choose the following uniform semantics. If bracketed output action c is annotated with the temporal interval (tmin, tmax), then action c will be performed no earlier than tmin units and no later than tmax units after the holding of its preset pre(c).

The standard verification algorithm for precedence constraints (described in section 4) can easily be extended to check these new delay constraints. When checking for a

(precedence) safety violation at system action σ, we determine whether there is a causal chain to σ from each member of σ's noncausal preset, say, pre(σ). First, copy the timing window on each output action to each of its predecessor arrows. Second, find the sums of tmin and tmax along all causal chains to σ from each member of its noncausal preset pre(σ). Consider the maximum delay case. For $\tau \in$ pre(σ), define D(τ, σ) as the maximum sum of tmax values along any causal chain from τ to σ. Then system action σ will be performed no later than max over τ of D(τ, σ) units after the holding of its noncausal preset pre(σ). For the minimum delay case, define d(τ, σ) as the maximum sum of tmin values, and take the min over τ of d(τ, σ); σ will be performed no earlier than this many units after the holding of its noncausal preset.

6 Conclusion

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 progress properties is quadratic in n, the number of DME elements. The number of system states grows exponentially with n. For example, when 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 compiler-independent 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 partially-ordered system computation is linear, because one constructs and compares simple presets. The limiting resource is the quadratic space used to store the termination table. To repeat, both space and time are quadratic, in this example, to verify a concurrent system with exponentially many states. Building up the actual partially-ordered system computations themselves is unnecessary; we work directly with the uncomposed behavior automata of the system components. We have also shown, at least in the simple case of output-delay-constrained reactive systems, that verifying temporal window constraints is barely more expensive than verifying precedence constraints. In general, the achievable efficiency of a real-time verification algorithm is a sensitive function of the precise abstraction of real time used in the model.

References

[1] D.L. Dill, "Trace theory for automatic hierarchical verification of speed-independent circuits", PhD 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, "Modeling concurrency with partial orders", Int. J. of Parallel Prog., Vol.15, No.1, February 1986, pp. 33-71.

[4] 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.

[5] 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).

[6] 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.

[7] 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 Lect. Notes in Comput. Sci., Springer Verlag, forthcoming.

[8] D. K. Probst and H. F. Li, "Partial-order model checking: A guide for the perplexed". In K. G. Larsen and A. Skou, (Eds.), Workshop on Computer-Aided Verification '91, Proceedings, Department of Mathematics and Computer Science, Aalborg University, Report IR-91-5, July 1991, pp. 405-416. Also Lect. Notes in Comput. Sci., Springer Verlag, forthcoming.

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

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