A Simple Modern Correctness Condition for a Space-Based High-Performance Multiprocessor [1]

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

Abstract - **A number of U.S. national programs, including space-based detection of ballistic missile launches, envisage putting significant computing power into space. Given sufficient progress in low-power VLSI, multichip-module packaging and liquid-cooling technologies, we will see design of high-performance multiprocessors for individual satellites. In very high speed implementations, performance depends critically on tolerating large latencies in interprocessor communication; without latency tolerance, performance is limited by the vastly differing time scales in processor and data-memory modules, including interconnect times. The modern approach to tolerating remote-communication cost in scalable, shared-memory multiprocessors is to (i) use a multithreaded architecture, and (ii) alter the semantics of shared memory slightly, at the price of forcing the programmer either to reason about program correctness in a relaxed consistency model or to agree to program in a constrained style. The literature on multiprocessor correctness conditions has become increasingly complex—and sometimes confusing—which may hinder its practical application. We propose a simple modern correctness condition for a high-performance, shared-memory multiprocessor; the correctness condition is based on a simple interface between the multiprocessor architecture and high-performance, shared-memory multiprocessor; the correctness condition is based on a simple interface between the multiprocessor architecture and the parallel programming system.**

Keywords: high-performance multiprocessor in space, scalable shared-memory multiprocessor, multithreading, relaxed consistency model, multiprocessor correctness condition, parallel programming model, programming/architecture interface, local acknowledgment protocol, nonsequential consistency, hybrid model.

1 Introduction

A number of U.S. national programs, including space-based detection of ballistic missile launches, envisage putting significant computing power into space. When enabling technologies such as (i) low-power VLSI, including cold chips, (ii) multichip-module packaging, and (iii) space-based liquid cooling for hot chips—are sufficiently mature, we will see design of

[1]This research was supported in part by NSERC operating grants A3363, A0921, strategic grant MEF0040121 and the NCE Micronet network. E-mail: probst@crim.ca and probst@vlsi.concordia.ca.

space-based liquid cooling for hot chips—are sufficiently mature, we will see design of high-performance multiprocessors for individual satellites. These machines will communicate with each other and with ground-based supercomputers—for example, to support massive real-time data acquisition, up to two terabytes of data per day. Very high speed implementations of multiprocessor architectures—that is, scalable machines with short processor cycles—that are built from clusters of interconnected processor and data-memory modules have a potential performance bottleneck; they *must* tolerate the large latencies in interprocessor communication (remote memory accesses may take anywhere from 100 to 1000 processor cycles) [2,11]. Building a shared-memory parallel programming system on top of a so-called "distributed-memory" multiprocessor architecture may motivate us—after as much latency as possible has been tolerated by multithreading—to alter the semantics of shared memory slightly, at the price of forcing the parallel programmer either to reason about program correctness in a relaxed consistency model or to agree to program in a constrained style. The literature on multiprocessor correctness conditions has become increasingly complex—and sometimes confusing—which may hinder its practical application [1,3,4,9–11]. One positive note is that most relaxed consistency models (actually, these are not models of multiprocessors at all but rather restrictions on an implementation or protocol, expressed at the level of the processor/memory interface) support sequential consistency for a special class of well-behaved, i.e., well-synchronized, parallel programs. We propose a simple modern correctness condition for a high-performance, shared-memory multiprocessor; the correctness condition is based on a simple interface between the multiprocessor architecture and the parallel programming system.

2 Memory Consistency Models

The conceptual model inherited from von Neumann machines with relatively few processors is that *all* memory reads and writes appear to be atomic (indivisible). In early multiprocessors, this "default" atomicity was implemented by the indivisible read/write memory cycle. Closely related to atomicity in the traditional model is the view that each operation in a (sequential) program thread appears to execute before the next operation in the thread is issued. The high cost of waiting for each stream operation to globally perform before the next stream instruction issues is well known [7,8]. The first attempt to define a multiprocessor correctness condition that would permit assertional reasoning (essentially, the continued existence of a program state space) and still be efficiently implementable was Lamport's sequential consistency [12]. Assertional reasoning would still be possible, Lamport wrote, provided that "the result of any execution [by the multiprocessor] is the same as if the operations of all the processors [had been] executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program". In simpler language, the program behaves as if the memory accesses of all the threads were interleaved—without destroying program order—and executed sequentially.

There are at least two problems with sequential consistency as a correctness condition. First, it has been widely misinterpreted to mean that one processor's update to a shared variable must be reflected in every other processor's *view* before the updating processor may issue another memory access, thereby confusing the correctness condition—sequential

consistency—with one particular low-performance implementation [7,9]. In this interpretation, all memory accesses to all shared variables are serializable even in the absence of any explicit synchronization and/or dependence information in the parallel program. Second, (sequential) program order is an extremely opaque tool for expressing the semantics of a parallel program decomposed into threads (instruction streams) [5,11]. This is what makes DASH release consistency so conservative from a parallel-programming point of view—questions such as, does this P protect this read? are occluded, and the program interpreter must respect the (sequential) programmer synchronization as written. In the DASH protocol, this is governed by a set of rules for the permissible orderings among acquires, releases and ordinary memory accesses [9]. "Thread partial order" consistency is more forward looking—as far as possible, all such questions about the *protects* relation on program operations are answered clearly by the parallel programmer and expressed as a partial order on synchronizing and ordinary memory accesses.

When a shared-memory parallel program is presented to the "program interpreter" of a so-called "distributed-memory" multiprocessor, the machine is not—in current practice, anyway—given an abstract program specification that defines the *dependence order* on the set of program operations. In concrete terms, the dependence order includes such necessary temporal precedences among program operations as uniprocessor control and data dependences (and antidependences), interprocessor data flow- and anti-dependences, and interprocessor control dependences (including necessary temporal precedences arising from an operation that depends on the restoration of an invariant). By definition, the dependence order is the limit for optimization—assuming sequential threads as the starting point—by the program interpreter; it defines the necessary temporal precedences among program operations that must appear to be observed in any correct execution of the parallel program. This dependence order is defined over the set of *all* program operations, including synchronizing and ordinary memory accesses, control-transfer instructions and register-only instructions.

In general, it is unrealistic to expect the programmer to articulate the program dependence order to the program interpreter; this may be more realistic at lower levels where a correctness proof of a critical subalgorithm can help the programmer to discover the dependence order [13]. In practice, what the program interpreter has to work with is (i) all compiler-discoverable local order, and (ii) all programmer-supplied local order, including explicit synchronization and other dependence information. To aid the parallel programming system, the high-level parallel programmer ought to do the following: (i) identify each P-type synchronizing operation (i.e., one that potentially blocks this thread), (ii) identify each V-type synchronizing operation (i.e., one that potentially unblocks some other thread), (iii) identify each PV-type synchronizing operation (i.e., one that potentially either blocks this thread or unblocks some other thread), and (iv) indicate, for each thread, the thread partial order that specifies the necessary temporal precedences among synchronizing and ordinary program operations in that thread. By so doing, the parallel programmer defines the *protects* relation on program operations. To focus on semantics, we defer the question of how thread partial orders and scopes of synchronization primitives are expressed in parallel programming languages [2,5,11]. Instead, we adopt the "fiction" that threads are presented to the program interpreter directly as partial orders. The interpreter respects the programmer synchronization as written, with "thread partial order" replacing "program order". The

set of rules for respecting programmer synchronization in our protocol is given in Section 4. Reference [14] is an elegant introduction to partial orders in parallel systems.

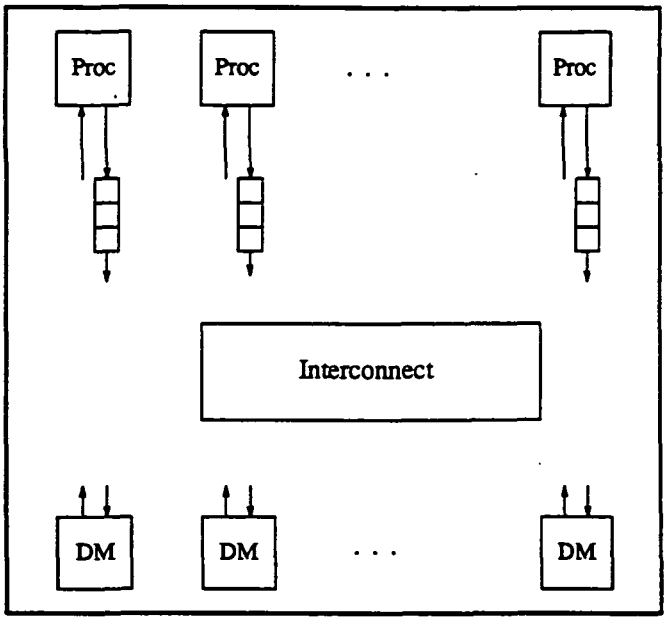

Figure 1: Abstract block diagram showing interconnection network, processor modules, data memory modules and operation buffers. No "dancehall configuration" is implied.

3 Thread Partial Order

Figure 1 shows an abstract block diagram for a multithreaded, scalable, shared-memory multiprocessor. Conceptually, the distinction between a distributed-memory and a shared-memory hardware architecture is small. In either case, any processor can access any address in the single, global address space. NUMA machines have a distinction between "cheap" local and "expensive" remote memory access, while UMA machines provide a constant-latency path between processors and memory. Multithreading means that a processor may issue a (possibly blocking—it depends entirely on the thread partial order) memory access by one thread via the operation buffer, and then—if necessary, i.e, if one has run out of concurrently-enabled operations in this stream—perform a context switch to some other thread while waiting for the first memory access to complete. Figure 1 applies equally well to architectures with various degrees of data caching, whether implemented in processor or data modules. Instruction caching is handled separately. The queues are operation buffers for memory accesses; these buffers contain multiple instructions that have been issued in parallel by processors and threads. Multithreading supports multiple pending operations by an individual processor. Relaxed consistency supports multiple pending operations by an

individual thread. Further optimization is possible when there are local acknowledgment protocols for memory accesses; this requires (data) caching of the processor's view [5].

The sharpest formulation of relaxed consistency is to state that memory accesses see "consistent" values *only* when such accesses are protected by synchronizing operations. Full consistency states that all memory accesses see consistent values (for example, reads return the most recent write) even when all synchronization is implicit. Most relaxed models include the programming constraint that every pair of conflicting memory accesses is protected by a synchronization chain. Two accesses *conflict* if they access the same location and at least one of them is a write. Abstractly, P-type operations define input synchronization points at which (i) partial or total program state is defined, and (ii) some invariants have been restored—because certain operations in other threads have completed. Similarly, V-type operations define output synchronization points at which (i) partial or total program state is defined, and (ii) some invariants have been restored—because certain operations in this thread have completed. PV-type operations (e.g., barriers) define both input and output synchronization points, and combine the semantics of P- and V-type operations [6].

When caches are present, processors (on which threads are scheduled) can have views. When local acknowledgment protocols are permitted due to (data) caching of a processor's view, "soft" V-type operations define *virtual* output synchronization points at which certain operations in this thread have sufficiently completed so that synchronization chains leaving this thread (possibly across processors) keep their usual semantics provided that we implement an additional protocol. Specifically, when semaphores are bound to the shared variables they protect, semaphore operations can be integrated with the remembering of updated values. The use of timestamps allows only the relevant portion of a processor's view to be copied to the other processor when there is a synchronization chain between threads on different processors [5]. Conceptually, this amounts to absorbing interprocessor communication into interprocessor synchronization.

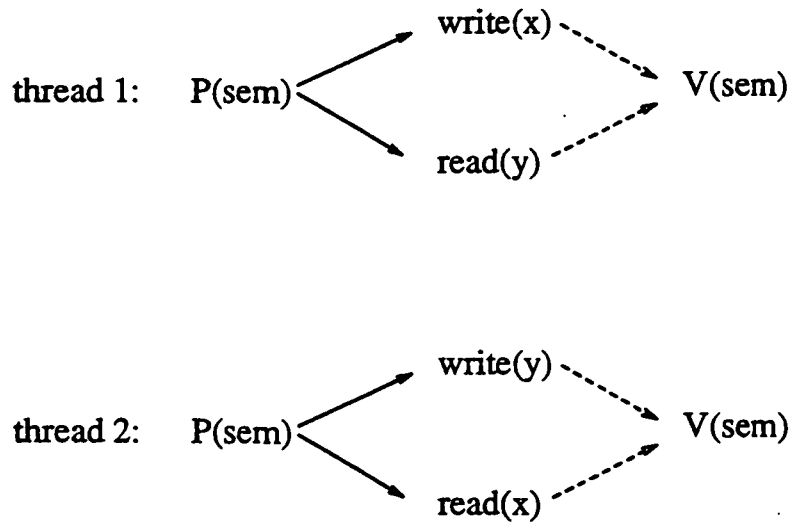

Figure 2: Two thread partial orders with local acknowledgment protocols.

Figure 2 shows the simple case of two critical sections in two threads where the operations in each critical section are concurrent. Here, the operations in each P/V-bracketed pair commute. When two concurrent operations in a thread partial order do not commute (i.e., when both orderings are permitted but have different results), a partial-order representation with branching structure may be required [15–17]. Incidentally, programmer-supplied local order is not restricted to defining the *protects* relation; it may also include specifying necessary temporal precedences between two program operations neither of which is a synchronizing operation (here, the theory becomes nontrivial). Without loss of generality, suppose that thread 1 has entered its critical section ahead of thread 2. At this point, there is a flow dependence from write(x) to read (x), and an antidependence from read(y) to write(y).

When there is no data cache, thread 1 must learn that both its memory accesses have completed with respect to thread 2 before its V operation may be issued. In this case, global acknowledgment protocols notify thread 1 of completion, while synchronizing operations guarantee completion to thread 2. When there is a data cache, thread 1 may issue its V operation as soon as local acknowledgment protocols for both its memory accesses have completed. Why? When thread 2 issues its P operation, an additional protocol can ensure that the P operation does not complete until (i) the new value of x has been communicated from thread 1 to thread 2, and (ii) the read of y has completed in thread 1 (see the next paragraph). This form of "delayed consistency" allows thread 1 to continue past its V operation before the output synchronization point has been actually reached [5].

The dashed arrow from read(y) to V(sem) in thread 1 may appear puzzling at first sight. Dashed arrows only make sense if the processors on which threads 1 and 2 are scheduled can remember their own views. In this case, why should there be an antidependence from read(y) to write(y)? If both threads are currently scheduled on the same processor, then they share the same view and the antidependence exists. If both threads are currently scheduled on different processors, then their views are disjoint and no antidependence exists. Since only the runtime system can determine which threads are currently on which processors, it is simpler for the compiler to generate code *as if* both threads currently shared the same view.

The graphics in Figure 2 can now be explained. In a thread partial order, a solid arrow $a \longrightarrow b$ means: b may not be issued until a has completed, while a dashed arrow $a \dashrightarrow b$ means: b may not be issued until a has *locally* completed (that is, has had its serialization order irrevocably determined). Abstractly, Figure 2 illustrates a 2-phase token release protocol. This is an implementation optimization; using only solid arrows in thread partial orders would not alter the correctness condition that was being implemented.

Figure 3 illustrates why the widespread emphasis on critical sections in discussions of relaxed consistency (due partly to programmer familiarity and partly to unexamined tradition, we suppose) is misleading. Here, there is a static dependence between a group of operations in one thread (protected by a V-type operation) and a group of operations in another thread (protected by a P-type operation). It seems strange to enforce a simple flow dependence between two threads by unlocking in one thread and locking in the other (although—to borrow a joke from Burton J. Smith—we could call this "generalized critical section" synchronization). The constraint issue is whether we still require that any thread's access to a shared variable be dynamically enclosed between a P-type and a V-type operation—as judged by

the thread partial order—even in cases where we are enforcing a static dependence. This constraint is satisfied in textbook solutions to the bounded-buffer problem, where both static and dynamic dependences are enforced. The answer in general is, it depends. If there is a dependence of these operations in thread 1 on "earlier" operations, then they *should* be bracketed, otherwise not.

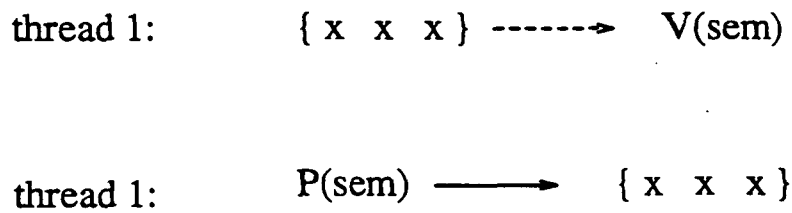

Figure 3: Synchronization primitive scope without critical sections.

4 Nonsequential Consistency

Sequential consistency and processor consistency are correctness conditions while weak consistency, release consistency, entry consistency, etc., are implementation restrictions [5,7,9]. Sequential consistency asserts that a multiprocessor execution is incorrect if it cannot be simulated by a serial (uniprocessor) execution that respects the "program order" of each thread. Serial program order is thus the test of whether the noncommutativity of two conflicting ordinary operations *matters*. The obvious modification is to replace "program order" by "thread partial order". The new condition reads as follows. First, all operations within a thread appear to execute in thread partial order, as viewed by the processor executing that thread. Second, the result of program execution is the same as if all synchronizing and ordinary memory accesses had been executed in some sequential order, and the projection of this sequence onto each thread is a linearization of that thread's partial order. Serial execution means possible (i.e., legal) execution on a uniprocessor. A multiprocessor satisfying this condition will be called *nonsequentially consistent*. But this is not quite right.

The role played by a programming constraint should be stated explicitly as a premise of the correctness condition; this is preferable to starting with an (arbitrary) implementation restriction and arguing that it implements a traditional correctness condition like sequential consistency given the programming constraint. In this spirit, coarsen the granularity to consider just the synchronizing and ordinary shared-variable accesses in a parallel program. We say that, if an ordinary shared-variable access is dynamically enclosed between a P-type and a V-type operation, then it is a *protected* access; otherwise, it is an *unprotected* access. A general correctness condition should include both types of access, and should allow for concurrent noncommutative operations within a single thread. If a parallel program is run on a nonsequentially-consistent multiprocessor, then—at this level of granularity—every correct execution on this machine has the following property:

(*i*) For each thread, there exists a serial execution of *all* the program operations that simulates the actual execution, and whose projection on that thread does not violate thread partial order.

(*ii*) Moreover, there exists a serial execution of *only* the synchronizing and protected accesses that simulates the matching portions of the actual execution, and—for each thread—the projection of this sequence on that thread does not violate thread partial order.

The easiest way to visualize the second condition is to imagine that the parallel program has been divided into properly- and improperly-synchronized *epochs*; when the final state of one epoch is intended as the initial state of another (this is the usual case), the two epochs will be separated by a barrier. During improperly-synchronized epochs, only processor consistency will hold. During properly-synchronized epochs, sequential consistency will hold. Serial simulation of a properly-synchronized epoch must take the initial and final states—which hold at the initial and final barriers, respectively—into consideration. In this way, a single, piecemeal serial execution can simulate all properly-synchronized portions of the actual execution.

We propose the following implementation of nonsequential consistency; temporarily nameless to avoid further confusion, it is one of many members of the "release consistency" family of implementation restrictions. The primary influence—apart from viewing threads as partial orders—is Tera release consistency [6,18] with an optional dash of entry consistency [5] for those people who believe that caches are the best way to implement shared memory (they are certainly not the *only* way).

(*i*) No processor may issue a shared-variable access until all P-type operations protecting it—that is, all P-type operations that precede the memory access in thread partial order—have completed.

(*ii*) No processor may issue a V-type operation until all shared-variable accesses protected by it—that is, all memory accesses that precede the V-type operation in thread partial order—have (at least locally) completed.

(*iii*) When a thread contains a PV-type operation (e.g., a barrier), all ancestor shared-variable accesses must have (at least locally) completed before the PV-type operation may issue, and the PV-type operation must have completed before any descendant shared-variable access may issue, where memory-access "ancestor" and "descendant" of a PV-type operation are defined by thread partial order.

That is, the program interpreter at each processor must run completion protocols for both synchronizing and ordinary memory accesses. When there are data caches available to processors, the optimization of "delayed consistency" is possible.

5 Conclusion

Discussions of correctness conditions and memory consistency models in shared-memory multiprocessors have grown needlessly complex in the last five years as implementation detail and excess formalism have obscured the simple language (i.e., semantics) issues. By cleanly separating the correctness condition (i.e., multiprocessor model) from the implementation

restriction (i.e., protocol model), we have brought out the simplicity of the "release consistency" implementation family. The key to simplicity is the programmer's definition of the *protects* relation on synchronizing and ordinary memory accesses. In the absence of full program dependence order, this is perhaps the most useful information that can be supplied by the programmer to the program interpreter; a good optimizing compiler can then adjust for uniprocessor dependences without altering the protects relation. Again, there will be elements of programmer-supplied local order in addition to the *protects* relation. We have proposed a new correctness condition which encompasses a wide range of parallel programming models, including ones with a mixture of shared-memory and message-passing semantics. The requirements specification of a *special-purpose* high-performance multiprocessor architecture for space applications would not include applicability to a wide spectrum of problems, but that does not make the ideas of this paper any less relevant (for example, they could be used to evolve the AT&T DSP3 Parallel Processor). We have indicated how to make an intelligent division of labor between programmer and optimizing compiler in discovering the thread partial order. One distinguishing feature of space applications is a concern for reliability; we are currently investigating a distributed-memory architecture for a *fault-tolerant* nonsequentially-consistent shared memory. Space systems are distributed systems, and—in this context—fault tolerance is as important as performance.

References

[1] S. Adve and M. Hill, *Weak ordering – a new definition*, in Proc. 17th International Symposium on Computer Architecture, May 1990, pp. 2–14.

[2] R. Alverson et al., *The Tera Computer System*, in Proc. 1990 International Conference on Supercomputing, June 1990, pp. 1–6.

[3] H. Attiya and J. Welch, *Sequential Consistency versus Linearizability*, in Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures, July 1991, pp. 304–315.

[4] H. Attiya and R. Friedman, *A Correctness Condition for High-Performance Multiprocessors*, in Proc. 24th ACM Symposium on the Theory of Computing, May 1992, pp. 679–690.

[5] B. Bershad and M. Zekauskas, *Midway: Shared-Memory Parallel Programming with Entry Consistency for Distributed-Memory Multiprocessors*, Department of Computer Science, Carnegie Mellon University, Report CMU-CS-91-170, April 1991.

[6] D. Callahan and B. Smith, *A Future-Based Parallel Language for a General-Purpose Highly-Parallel Computer*, in D. Gelernter et al. (Eds.), Languages and Compilers for Parallel Computing, MIT Press, 1990, pp. 95–113.

[7] M. Dubois et al., *Memory Access Buffering in Multiprocessors*, in Proc. 13th International Symposium on Computer Architecture, June 1986, pp. 434–442.

[8] M. Dubois and C. Scheurich, *Memory Access Dependencies in Shared-Memory Multiprocessors*, IEEE Transactions on Software Engineering, **16:6**, June 1990, pp. 660–673.

[9] K. Gharachorloo et al., *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors*, in Proc. 17th International Symposium on Computer Architecture, May 1990, pp. 15–26.

[10] P. Gibbons et al., *Proving Sequential Consistency of High-Performance Shared Memories*, in Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures, July 1991, pp. 292–303.

[11] P. Gibbons and M. Merritt, *Specifying Nonblocking Shared Memories*, in Proc. 4th ACM Symposium on Parallel Algorithms and Architectures, June 1992.

[12] L. Lamport, *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, **28:9**, September 1979, pp. 690–691.

[13] L. Lamport, *How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor*, preliminary draft, May 1992.

[14] V. Pratt, *Modeling concurrency with partial orders*, Int. Journal of Parallel Prog., **15:1**, February 1986, pp. 33–71.

[15] D. Probst and H. Li, *Using partial-order semantics to avoid the state explosion problem in asynchronous systems*, in E. Clarke and R. Kurshan, (Eds.), Workshop on Computer-Aided Verification '90, June 1990, DIMACS Series, Vol. 3, 1991, pp. 15–24. Also Lecture Notes in Computer Science 531, Springer–Verlag, 1991, pp. 146–155.

[16] D. Probst and H. Li, *Partial-order model checking: A guide for the perplexed*, in K. 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 Lecture Notes in Computer Science 575, Springer–Verlag, 1992, pp. 322–331.

[17] D. Probst and L. Jensen, *Controlling state explosion during automatic verification of delay-insensitive and delay-constrained VLSI systems using the POM verifier*, in S. Whitaker, (Ed.), Proceedings of the 3rd NASA Symposium on VLSI Design, Moscow, ID, October 1991, pp. 8.2.1–8.2.8.

[18] B. Smith, personal correspondence.