PARALLEL COMPOSITION OF LOCKSTEP SYNCHRONOUS PROCESSES FOR HARDWARE VALIDATION: DIVIDE-AND-CONQUER COMPOSITION

Ganesh C. Gopalakrishnan[1]
Dept. of Computer Science, University of Calgary,
Calgary, Alberta, Canada T2N1N4
email: ganesh@cs.utah.edu
Narayana S. Mani, and **Venkatesh Akella**
Dept. of Computer Science, Univ. of Utah,
Salt Lake City, UT 84112, USA

Abstract

Consider a process M implemented as a collection of subprocesses SM_i. To certify the implementation to be correct, the collective behaviors of SM_i and the behavior of M are compared using a suitable verification criterion. In many approaches the implementation is specified *structurally* using operators such as ||, hiding, and renaming while M is specified *behaviorally* using *action prefixing* and *choice* operators. This style is being used for hardware specification also [10, 8, 4, 3]. In this paper we address the question whether behavioral specifications can be deduced rapidly from structural specifications in the setting of a simple language HOP. We also address the question of doing the same for geometrically regular (array) structures that abound in VLSI. We present two algorithms PARCOMP, and PARCOMP-DC, report their performance, and explain the heuristics used to make them efficient.

1 Introduction

Composition and abstraction are central to design. Composition goes hand-in-hand with abstraction—the act of retaining only the *observable behavior* and not the *internal structure*. Here are some real-world examples: the connection of two resistors in parallel and the computation of the equivalent resistance; the conjunction of two logical formulae followed by the existential quantification of a common variable [5]; composition of processes (*e.g.* [6] [12]); parallel composition of transistor networks (*e.g.* [7], [8]).

We present a language, called 'HOP' (Hardware viewed as Objects and Processes) for specifying synchronous hardware systems as well as their implementations. In HOP, hardware systems are modeled as finite state transition systems (*i.e.* the number of control states are finite; data states may be unbounded). The main topic of this paper is an efficient algorithm for process composition called PARCOMP, that infers an abstract (in the above sense) behavioral description M' from a collection of HOP processes SM_i. This algorithm is used to facilitate formal verification, symbolic simulation as well as ordinary simulation; in fact, we are lead to believe-in and pursue an approach where verification and simulation are two points in a continuum of validation related activities, [3]. We now touch upon some features of PARCOMP.

PARCOMP has been implemented as a part of the 'HOP system' in Common Lisp, and found to run with practically acceptable speeds on all the examples tried so far (the largest to date being

[1]On 1-year leave from University of Utah. Supported in part by NSF Award MIP-8710874

the cache memory system shown in figure 1). Heuristics responsible for the speed of PARCOMP are consistent with HOP's operational semantics [4]. PARCOMP is being used as a preprocessor for simulation (we obtain M', simplify it using algebraic rewrite rules to get M'', and then simulate M'') [9]. PARCOMP has also been adapted to work efficiently for geometrically regular arrays that occur commonly in VLSI *e.g.* figure 2. Such arrays are more general than systolic arrays because they needn't be computationally regular, and can have global busses embedded within them. We call such arrays *arhythmic arrays*. This version of PARCOMP, called PARCOMP-DC, uses a divide-and-conquer technique to effect considerable savings in run-time in inferring the behavior of arhythmic arrays.

Main Results

Suppose we are given a process definition Hide e in $P \parallel Q$ where 'Hide' and $\parallel$ are operators in HOP that are similar in purpose to the corresponding operators in CSP[6] or SCCS[13]. Suppose PARCOMP is to deduce an equivalent single process R. We show that it is several orders of magnitude faster if PARCOMP uses information on hidden actions *during* the application of the rules for the $\parallel$ operator, rather than after having applied all such rules.

PARCOMP-DC exploits the facts that: (i) $\parallel$ is commutative and associative; (ii) the cells of arhythmic arrays are identical except for the names of ports and events. It splits the array into two halves and computes the behavior of only the 'left-half'. It then obtains the behavior of the 'right-half' through a *renaming* operation. This process is recursively applied to the left-half of the left-half, and so on. In addition, PARCOMP-DC condenses the inferred behavior by using functional constructors that employ *universal quantification* over array indices to succinctly express the inferred behavior. An added advantage of employing universal quantification is that it immediately reveals general properties enjoyed by the array.

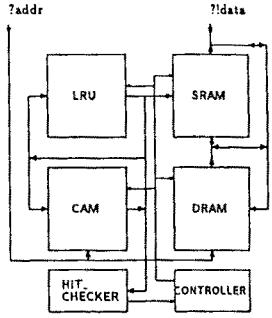

Figure 1: A Cache Memory System

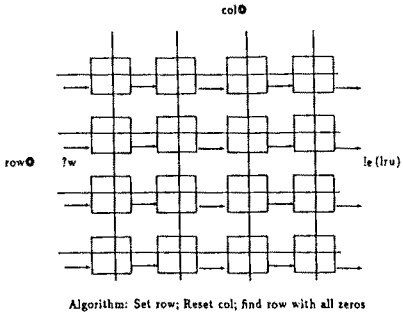

Figure 2: An LRU Unit

1.1 The Language HOP

Each HOP specification of a module specifies the *interface* of the module, which consists of: (a) *ports*: the wires that constitute data I/O ports; (b) *events*: the wires that bring in control commands and take back status results; (c) the *protocol* of usage. The protocol is specified in a simple process specification notation whose semantics can be captured using a (finite control state) labeled transition system [14].

The *instantaneous description* (ID) of a HOP process consists of its control and data-part states. (Data-part state denotes that component of a module's state that has very little effect on the control flow through control states; e.g. the contents of a RAM memory has little effect on the control protocol of the memory. Processes move from one ID to another through a set of *observable actions*. We write

$$Control\_state[Datapart\_state] \stackrel{Observable\_Actions}{\longrightarrow} New\_Control\_State[New\_Datapart\_State].$$

Control states are uninterpreted atoms. Datapart states are values of data types such as *stack*, *tree*, *array*, *byte*, etc. Datapart state elements need not be bounded in size (except for practical reasons). The transitions of processes are caused by clock beats. Transitions are labeled (= annotated) by (possibly empty) sets of actions, one set per action variety presented below:

1. *Input events*, which are shown as annotations on transitions. An input event 'e' is written 'Ie'.
2. *Data query*: "x=?p" means "let x be the value (of some data type) coming through input port p at the time the data query is evaluated". 'x' can be assigned only once, as it stands for a particular value, and not a memory location.
3. *Boolean guards*: Similar to the boolean guards of CSP[6], HOP's boolean guards are first order formulae of type *boolean*.
4. *Output events* are similar to input events, and written *e.g.* 'Oe'.
5. *Data assertion*: "!q=E" means that the expression "E" is evaluated and asserted on output port q. (Expressions are defined in a first-order functional sublanguage. The use of user-defined abstract data types is also supported.)

The meaning of these annotations is straightforward: Input events labeling a transition must be true for the transition to be taken. Boolean guards help express the dependencies of control flow on data values, and, in conjunction with input events, determine whether a transition is taken or not. When a transition is taken, the output events labeling it are asserted, data queries are performed, data assertions are made, and the internal data part state is updated. Any of these categories of annotations may be empty. If no input events are shown, the transition is unconditionally taken.

Every HOP process can be described at the level of *abstract syntax* via a collection of *process equations*. One process equation is of the form:

$$P[vars] = |_i\ ie_i, dq_i, g_i\ :\ oe_i, da_i \rightarrow P_i[Exp_i]$$

and says that process $P[vars]$ has $i \in I$ moves. The ith move takes place if all the input events in the set ie_i are offered, the data queries in the set dq_i are performed, and the guard expression g_i (which may contain the variables in *vars* and those in dq_i) is true. Then, the set of output events oe_i and the set of data assertions da_i are asserted, and the next ID (= active process) is $P_i[Exp_i]$.

1.2 Comparison with Related Work

Lustre[2] and Esterel[1] are two languages that resemble HOP in terms of lockstep synchronous execution. Milne [11] presents parallel composition as an approach to simulation and verification in the setting of CIRCAL. In the HOL system [5] there is an inference rule called the 'existential

$$
\begin{aligned}
P[vars] ::= & \ |_i ie_i, dq_i, g_i \ : \ oe_i, da_i \rightarrow P_i[Exp_i] \\
& \mid P_1[Exps_1] \parallel P_2[Exps_2] \\
& \mid \text{Hide ie in } P[Exp] \ \mid \text{Hide oe in } P[Exp] \\
& \mid \text{Hide ?p in } P[Exp] \ \mid \text{Hide !p in } P[Exp]
\end{aligned}
$$

Figure 3: Abstract Syntax of HOP

quantification elimination' that achieves the same end as PARCOMP, albeit with several manual steps.

Our work may be distinguished in the following ways. The ordinary PARCOMP has been applied for several large examples, and we propose two heuristics for speeding up the composition: *Lockstep Cartesian Product* computation, and *early pruning based on hidden events.* We have not seen these two heuristics reported in other works. We have not come across any algorithm similar to PARCOMP-DC that exploits the regularity of geometrically regular VLSI arrays.

2 The Syntax and Semantics of HOP

Descriptions written in the user level syntax of HOP (not presented here, but see [3, 4]) can be translated into descriptions in the *abstract syntax* of HOP, whose grammar is given in figure 3.

Rule for Deterministic Choices

This rule simply says that every choice defines a possible behavior:

$$(|_i \ ie_i, dq_i, g_i \ : \ oe_i, da_i \rightarrow P_i[E_i]) \ \stackrel{ie_i, dq_i, g_i \ : \ oe_i, da_i}{\longrightarrow} \ P_i[E_i]$$

Rule for Parallel Composition

This rule computes the possible behaviors of $P[v1] \parallel Q[v2]$ from those of $P[v1]$ and $Q[v2]$.

$$\frac{P[v_1] \ \stackrel{ie1, dq1, g1 \ : \ oe1, da1}{\longrightarrow} \ P'[E_1], \ Q[v_2] \ \stackrel{ie2, dq2, g2 \ : \ oe2, da2}{\longrightarrow} \ Q'[E_2]}{P[v_1] \parallel Q[v_2] \ \stackrel{\substack{IE(ie1 \cup ie2, \ oe1 \cup oe2), \\ DQ(dq1 \cup dq2, \ da1 \cup da2), \\ G(dq1 \cup dq2, \ da1 \cup da2, \ g1 \wedge g2) \ : \\ (oe1 \cup oe2), \ (da1 \cup da2)}}{\longrightarrow} \ P'[E_1'] \parallel Q'[E_2']}$$

Here, the $\cup$ operation takes the set union of its arguments. The helping functions employed above are now defined.

$IE(ie, oe) = removetag(ie) \setminus removetag(oe)$ where $removetag$ strips the direction tags 'I' and 'O'. Informally, those ie that are not synchronized by matching oe are left behind. $DQ(dq, da)$ returns every $(q = ?p) \in dq$ for which there is no corresponding $(!p = E) \in da$. We retain data queries that are not matched by corresponding data assertions. $G(dq, da, g) = instantiate(g, \ bindings(dq, da))$ where $bindings(dq, da) =$ set of (var, exp) such that for every $(var = ?p) \in dq$ there is a corresponding $(!p = exp) \in da$. In this step, we first determine the variable bindings that result from having simultaneous data assertions and queries on the same port. These variable bindings are then used to instantiate guard expressions. Thus we are simulating the effect of value communications among processes symbolically. $E_1' = instantiate(E_1, \ bindings(dq, da))$; and $E_2' = instantiate(E_2, \ bindings(dq, da))$. These take into account how the data part state of the processes change as a result of value communications between processes.

Rules for Hiding

These simple rules capture what can be ignored as a result of internalizing events and ports. We first consider the most practically important of all these rules—the 'Hide ie in P' rule. This rule says that hiding an input event causes the choice arm guarded by that input event to be dropped. The key idea behind this rule is to "distill away" behaviors that will not materialize at each point in time:

$$\frac{P[v] \xrightarrow{ie1,dq1,g1\,:\,oe1,da1} P_1[E_1],\ ie \notin ie_1}{Hide\ ie\ in\ P[v] \xrightarrow{ie1,dq1,g1\,:\,oe1,da1} Hide\ ie\ in\ P_1[E_1]}$$

Hiding an output event oe merely suppresses this assertion from the outside world; no computational paths are pruned:

$$\frac{P[v] \xrightarrow{ie1,dq1,g1\,:\,oe1,da1} P_1[E_1],\ oe \in oe1}{Hide\ oe\ in\ P[v] \xrightarrow{ie1,dq1,g1\,:\,oe1 \setminus oe,\ da1} Hide\ oe\ in\ P_1[E_1]}$$

If ?p is an input port, and if a data query $x = ?p$ is made through ?p, then hiding ?p from a process P prevents P from accepting inputs via this port. We simply take away the data query, and so $x \in g1, da1, E_1$ will remain unbound. This may be okay if the value of x need not be known in evaluating $g1, da1$ and E_1:

$$\frac{P[v] \xrightarrow{ie1,dq1,g1\,:\,oe1,da1} P_1[E_1],\ (x = ?p) \in dq1}{Hide\ ?p\ in\ P[v] \xrightarrow{ie1,\ dq1 \setminus (x=?p),\ g1\,:\,oe1,da1} Hide\ ?p\ in\ P_1[E_1]}$$

Hiding an output port is similar to hiding an output event. All data assertions made on port !p are expunged when port !p is hidden:

$$\frac{P[v] \xrightarrow{ie1,dq1,g1\,:\,oe1,da1} P_1[E_1],\ (!p = E) \in da1}{Hide\ !p\ in\ P[v] \xrightarrow{ie1,dq1,g1\,:\,oe1,\ da1 \setminus (!p=E)} Hide\ !p\ in\ P_1[E_1]}$$

The recursive application of the hiding rule—$Hide\ !p\ in\ P_1[E_1]$ for example—captures how PARCOMP effects the hiding rule as it unravels the timing behavior of the processes.

The intended semantics of HOP is that of deterministic execution. Sufficient syntactic conditions that guarantee determinacy have been reported in [4].

3 A Sketch of PARCOMP

3.1 Lock-step Cartesian-product Process

Basically, PARCOMP computes (what we define to be) the *lock-step cartesian product* of two transition systems. Given two process graphs A and B, a lock-step cartesian product (LCP) of A and B, written $lcp(A, B)$, is obtained by applying the following steps until no new states or edges are added:

1. If A_0 is the initial state of A, and B_0 is the initial state of B, then the pair $< A_0, B_0 >$ is in $lcp(A, B)$.

2. If state $< A_i, B_i >$ is in $lcp(A, B)$, and there is a directed edge E_{ij} going from A_i to a state A_j in A, (and likewise F_{ij} is a directed edge going from state B_i to a state B_j in B), then $< A_j, B_j >$ is added to $lcp(A, B)$. Further, the edge EF_{ij} is added to $lcp(A, B)$ directed from $< A_i, B_i >$ to $< A_j, B_j >$. EF_{ij} is labeled with annotations computed from those on E_{ij} and F_{ij} using the rule for $\|$.

   The following optimization makes PARCOMP a practically efficient algorithm:

3. After the above step, if the transition is labeled by some input event e that is hidden, then drop this LCP edge, and do not generate that portion of the LCP reachable via this edge. This step uses the rule for hiding input events.

   The following check often reveals sequencing errors:

4. Every dead-end state in the LCP is indicative of the presence of sequencing errors in the specification of one or more of the subprocesses SM_i. (We assume that none of the SM_i themselves had dead-end states.) A dead-end state is formed thus: a state ends up having all exits from it through transitions that have at least one unsynchronized and hidden input event; these transitions are pruned by step 3.

We compute the lockstep cartesian product and not the traditional cartesian product (which has more states) because only configurations in LCP are reachable due to 'the marching in unison with the beats of a clock'. This is the first level of optimization.

Step 3 corresponds to 'distilling away' modes of behavior that are not implemented. For example, if a submodule can perform multiplication, but within a system it is never asked to multiply, then the capability to multiply can be discarded. More importantly if a module can perform multiplication but is not asked to multiply at time t, then those control states reachable at time t will not offer a choice corresponding to the multiply operation. This heuristic brings about anywhere from 10 to 100 times speed-up. Step 4 has proven to be quite valuable in debugging complex specifications, especially the pipelined Cache memory system of figure 1. Several examples of PARCOMP in action have been presented in [3, 4].

4 Run-times for PARCOMP

Let each transition be labeled with annotations, each of which are K long. Let T transitions be traversed in computing the LCP, and let there be N processes altogether. In taking one LCP transition, PARCOMP clashes N, K-long annotations. This costs K^N because any action of one annotation can interact with an action of any other annotation. Since there are T transitions, we get a total run-time of $O(T.K^N)$. In many classes of designs, however, T determines the run-time with N remaining relatively fixed (less than 10 in many cases). In such cases the run-time is linear in T (which can be very large). Early pruning based on unsynchronized events can dramatically reduce T. Some figures of run-time are shown in figure 4.

EXAMPLE	Ctrl States In Cartesian Prod.	Ctrl States In Final Process	Transitions Pruned During PARCOMP	Transitions In Final Output	Run time (secs) KCL, 16M Sun 3/50
Pipelined Stack	24	12	220	21	1.97
Clock Shaper	2	2	0	5	0.32
Huffman Encoder	2	2	15	6	0.95
Feedback Shift Reg. Counter	1	1	105	4	0.82
Pipelined Cache mem.	98	15	13,413	28	328

Figure 4: Performance of PARCOMP on Five Examples

5 PARCOMP-DC: Basics, and Performance

Consider the array A shown in figure 5. It consists of a collection of modules M connected in a regular interconnection pattern. For simplicity of explanation, assume a nearest-neighbor connection that is regular in both the dimensions.

Consider the problem of computing $PARCOMP(A)$; *i.e.* the composition of all the Ms constituting A. $PARCOMP$ is both commutative and associative. Hence, we can split A into two halves, say A_T standing for "the top of A" and A_B, standing for "the bottom of A", and obtain:

$$PARCOMP(A) = PARCOMP(\ PARCOMP(A_T),\ PARCOMP(A_B)\ ).$$

Since A_T and A_B differ only in the names of their external ports, we need compute *only* $PARCOMP(A_T$ $PARCOMP(A_B)$ can be obtained from this by subjecting $PARCOMP(A_T)$ to *renaming* operations. Thus PARCOMP-DC does *log N* ordinary PARCOMPs of two modules at a time.

In many real-world arrays, the transition system of the whole array has the same number of control states as the transition system of its basic cell. *E.g.*: In figure 2, each cell does something during clock-high and something else during clock-low—all on single-bit quantities. The whole array does similar actions, but on vectors, instead of single bits. In such cases, the run-time of the PARCOMPs done during PARCOMP-DC is determined solely by the step of clashing the actions labeling the transitions. In other words, PARCOMP-DC clashes *two* processes at a time; *log N* such clashings are done altogether between pairs of processes. Each time around, the Ks are bigger than before.

The very first time, both processes have K-long annotations. After clashing once, each annotation grows to at most $2K$. This is because: (a) in the worst case all events are distinct, and so we simply pool them; (b) for events that synchronize, the input event is discarded; (c) for all data assertion/query pairs that communicate, the assertion-expression is substituted in place of the query-variable, and the query is discarded; (d) data assertion/query pairs that do not communicate simply are pooled together.

Thus, we sum the costs for each of the *log N* steps. Since the cost of clashing annotations ($O(K^2)$) dominates the cost of copying and renaming process descriptions ($O(K)$), only the former cost is summed:

$$\begin{aligned} & T.(K^2 + (2K)^2 + (4K)^2 + \ldots \texttt{ log N terms} \\ = \ & TK^2.((2^0)^2 + (2^1)^2 + (2^2)^2 + \ldots + (2^{(log\ N)-1})^2 \\ = \ & O((2^{(log\ N)-1})^2) \\ = \ & O(N^2) \end{aligned}$$

If we assume a fixed K, then PARCOMP gives exponential growth of time with N, but PARCOMP-DC gives only polynomial growth. If N is fixed, then both are polynomial, but PARCOMP has a higher degree for $N > 2$.

We have run PARCOMP-DC on the LRU matrix for various values of N, from 4 to 256 (*i.e.* using arrays with sizes 2x2 upto 16x16) (figure 6). For this array, the run-time is better (quicker) than $O(N^2)$.

6 Concluding Remarks

Several verification techniques can be supported by PARCOMP and PARCOMP-DC. One specific approach that we have pursued [3] consists of the following steps: (a) obtain the inferred behavior; (b) embody the verification criterion in a *tester process*—a HOP process capable of exercising the module under test in all legal ways; (c) compose the tester and the testee, and verify this composition.

Acknowledgements We are grateful to professor C.A.R. Hoare for suggesting improvements to the formulation of HOP's semantics during his visit to Utah. Thanks to the verification group at Calgary for providing feedback.

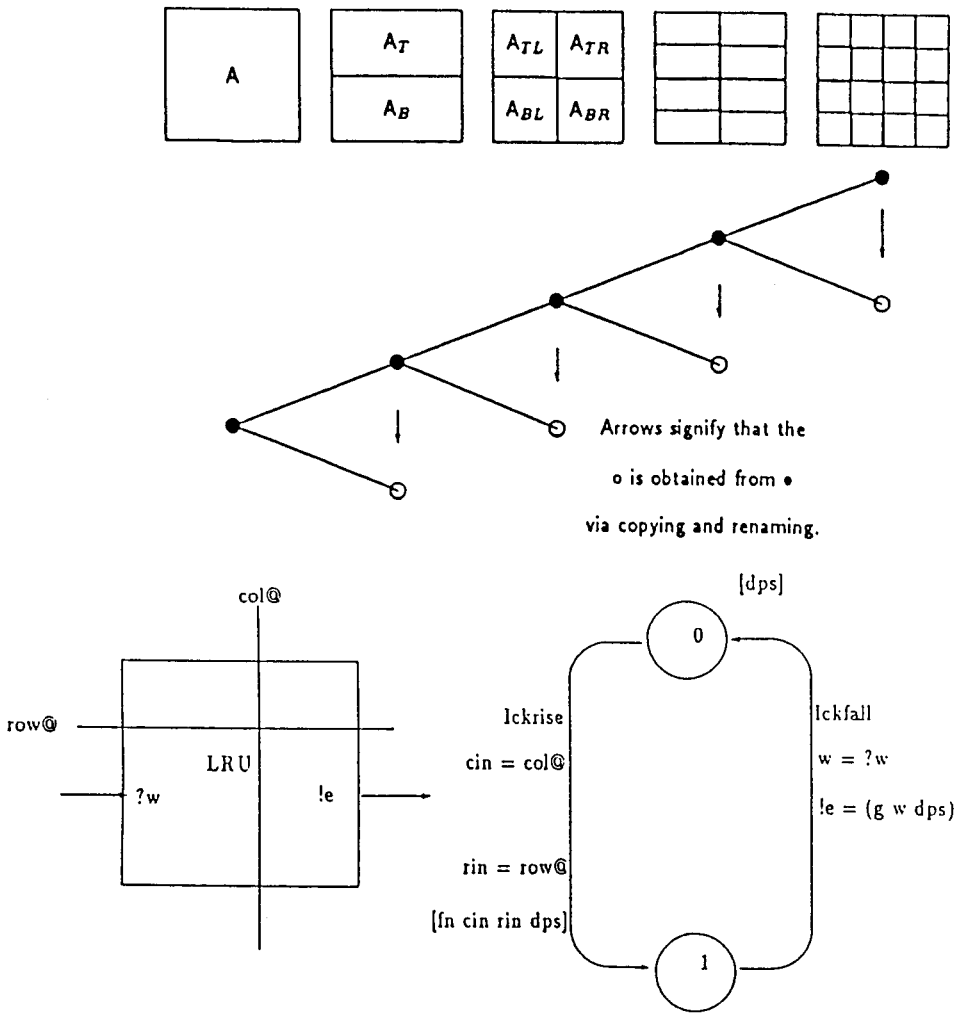

Figure 5: Divide and Conquer PARCOMP, and LRU Matrix Details

LRU Matrix Size	Run time (secs) KCL, 16M Sun 3/50
2 × 2	0.7
2 × 4	1.7
2 × 8	3.6
4 × 4	3.8
2 × 16	20.9
4 × 8	20.8
4 × 16	47.9
8 × 8	48.4
8 × 16	156.8
16 × 16	588.8

Figure 6: PARCOMP-DC Timings on Different Sizes of LRU Matrix

References

[1] Gerard Berry and Laurent Cousserat. The ESTEREL synchronous programming language and its mathematical semantics. In S.D.Brookes, A.W.Roscoe, and G.Winskel, editors, *Seminar on Concurrency, LNCS 197*, pages 389–448. Springer-Verlag, 1984.

[2] P. Caspi, D.Pilaud, N.Halbwachs, and J.A.Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, pages 178–188. ACM, 1987.

[3] Ganesh C. Gopalakrishnan. Specification and verification of pipelined hardware in HOP. In *Proc. Ninth International Symposium on Computer Hardware Description Languages*, pages 117–131, 1989.

[4] Ganesh C. Gopalakrishnan, Richard Fujimoto, Venkatesh Akella, and Narayana Mani. HOP: A process model for synchronous hardware. semantics, and experiments in process composition. *Integration: The VLSI Journal*, 1989. *Accepted for publication.*

[5] Michael Gordon. HOL: A proof generating system for Higher Order Logic. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

[6] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[7] C.A.R. Hoare. Formal methods in computer science—a case study, April 1989. Invited talk given in the Third Annual Organick Memorial Lecture Series, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112.

[8] C.A.R. Hoare and Michael J.C. Gordon. Partial Correctness of C-MOS Switching Circuits: An Exercise in Applied Logic. In *Proceedings of the Third Annual Symposium on Logic in Computing Sciences*, 1988. ISBN 0-8186-0853-6.

[9] Narayana Mani. Behavioral simulation from high level specifications. Master's thesis, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, May 1989. *MS Thesis, scheduled to be defended on May 31, 1989.*

[10] George G. Milne and Mauro Pezze. Typed circal: A high level framework for hardware verification. In *Proc. 1988 IFIP WG 10.2 International Working Conference on "The Fusion of Hardware Design and Verification", Univ. of Strathclyde, Glasgow, Scotland*, pages 115–136, July 1988.

[11] George J. Milne. Simulation and Verification: Related techniques for hardware analysis. In *Proceedings of the Seventh International Conference on Computer Hardware Description Languages*, pages 404–417. North-Holland, 1985.

[12] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980. LNCS 92.

[13] Robin Milner. Calculii for synchrony and asynchrony. Technical Report CSR-104-82, Univ. of Edinburg, 1982. Internal Report.

[14] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, September 1981.