A Compositional Model For Synchronous VLSI Systems

Ganesh C. Gopalakrishnan

Tech. Report. UUCS-87-024

A Compositional Model for Synchronous VLSI Systems

Ganesh C. Gopalakrishnan

Department of Computer Science, Univ. of Utah, Salt Lake City, Utah 84112
(801) 581-3568; ganesh@cs.utah.edu

September 9, 1987

Contents

1 Introduction **1**

2 Illustration of HOP **3**
- 2.1 An XOR gate . . . 3
- 2.2 A Two-phase Clocked Master-slave Flip-flop . . . 5
- 2.3 A Two Bit Maximal Length Sequence (MLS) Counter . . . 5
  - 2.3.1 Expressing the MLS Counter as a Process Realization . . . 6
  - 2.3.2 Lisp Representations of HOP Specifications . . . 7

3 Overview of the Semantic Basis of HOP and PARCOMP **7**

4 Illustration of PARCOMP **8**
- 4.1 Illustration of PARCOMP on the MLS Counter . . . 8
- 4.2 Results of the Run from Our Implementation . . . 9
- 4.3 Statically Detecting Synchronization Failures . . . 9

5 Work in Progress **9**

A Appendix **12**
- A.1 The Abstract Syntax of HOP . . . 12
- A.2 The Formal Operational Semantics of HOP . . . 12
- A.3 The Computational Model Underlying HOP . . . 15
- A.4 The Parallel Composition Algorithm . . . 15
- A.5 Some Details of Work in Progress . . . 17
  - A.5.1 Symbolic Simulation . . . 17
- A.6 Lisp Representations of HOP Specifications . . . 17
- A.7 One More Example: A Stack . . . 20
- A.8 A Pipelined Stack Controller . . . 20

List of Figures

1 The Absproc Specification of an XOR Gate . . . 3
2 The HOP Processes XOR, MUX, FF, CTRLR and MLS . . . 4
3 The Circuit and Absproc Spec. of a Two-phase Clocked MS FF . . . 5
4 An MLS Counter's Realization . . . 6
5 A Realproc Specification of an MLS Counter . . . 6
6 A Tester Process for an MLS Counter, Expressed in HOP . . . 10
7 Parallel Composition of a MLS Counter and Its Tester . . . 10
8 Abstract Syntax of HOP, with terminals in teletype font . . . 12
9 Definition of *Action Product* in HOP . . . 13
10 Operational Semantics of HOP . . . 14
11 Use of Data Assertions . . . 15

12 Realization of a Stack 20
13 *STKPAR* Obtained via Parallel Composition 20
14 Realization of a Stack 21

1 Introduction

Currently available hardware specification languages have two serious deficiencies: (i) inadequate protocol definition capabilities; (ii) lack of a compositional model. We now explain these in more detail.

1. *Lack of an Interface Protocol Definition Capability:* A concise and complete description of the usage protocols of modules cannot be stated in those hardware specification languages that we have come across. Some illustrative scenarios that aren't adequately handled are: (i) "Consider a two-phase clocked register. The register can be requested to perform *load* during phase-a; during the same phase-a, the register's output still reflects its old state (it is master/slave); during phase-b, it delivers a value equal to the loaded value". (ii) "A domino logic stage provides valid output during a certain phase and this output is destroyed due to precharging during the next phase". With these and other examples in mind, we make two specific observations:

   (a) *Events* (such as control lines attaining certain combinations at certain phases, etc.) as well as *synchronization skeletons/protocols* are important notions in VLSI specification; however most hardware specification languages (*e.g.* [Bar81], [Cam84], [Joh84], [She85], [CGM86]) don't support these notions well.

   (b) Values delivered on data ports depend upon states as well as input values. It is desirable to be able to treat complex *circuit modules* encompassing large amounts of state—such as a stack or a CAM—as *modules* possessing a single high-level state *at the modeling level as well*, rather than insisting (*e.g.* as most of the above hardware specification languages do) that every storage unit be bared. Those languages that insist that every state and combinational unit be *separated* as well as *bared* have two shortcomings:

      i. They provide very low-level "next state" and "output" equations (maybe in a functional or logic style);

      ii. Often such a separation is not satisfactory (a domino stage is a module with "storage" but usually implements boolean functions).

   Our solutions (elaborated later) are: For addressing point 1a, we have designed a language "HOP[1]" with emphasis on behavior and protocol specification; For point 1b, we propose the use of abstract data types to model the states and port values resulting from them. In this way, we can write "state equations" as well as "output equations" in terms of larger and more meaningful state entities. Our past research ([GSS86], [GS87], [Gop86]) supports this observation, in addition to some large examples recently specified by us ([FTG88]).

2. *Lack of an Underlying Compositional Model:* A specification language L is defined to provide a compositional model for the circuits it describes if, for an associative and commutative (AC) circuit composition operator '$\circ$' and an AC specification composition operator '$||$' in L, and a mapping Beh that yields the syntactic behavioral description of a given circuit as expressed in L, and a suitable behavioral equivalence relation $\equiv$:

   $$Beh\ (C_1 \circ C_2) \equiv ((Beh\ (C_1)\ ||\ Beh\ (C_2))$$

   The availability of such a compositional model opens up several new ways of attacking familiar but hard VLSI design problems, as we show in this paper.

[1] Hardware viewed as Objects and Processes

In this paper,

1. We present a specification language called HOP that, we hope to show, provides an adequate (in terms of rigor and practical appeal) protocol and behavioral specification capability for specifying modern VLSI. HOP borrows many ideas from Milner's Synchronous Calculus of Communicating Systems [Mil82]. We show that directly applying any of the existing Calculii of Communicating Systems (including SCCS) leads to problems in modeling communication through data busses. We introduce a new primitive for process interactions called *data assertions* to solve these problems.

HOP is vastly more simple than SCCS. HOP can describe only lockstep-synchronous circuits with deterministic behaviors under the *conservative clocking* [MC80] assumption. Within this framework, HOP boasts some new capabilities. Each HOP specification has two distinct sections: a *timing protocol section* written in a simple process specification notation, and a *functional behavior section* written in a first order functional programming language (that may use abstract data types). HOP specifications have so far proved to be easy and natural to write (and read) to us. Currently HOP is being used to specify an IC as complex as a Memory Management Unit that we are currently designing [FTG88].

2. We present an algorithm PARCOMP. With it, the above equation for compositionality becomes:

$$Beh\,(C_1 \circ C_2) \equiv \mathsf{PARCOMP}\,(Rep(\circ), Beh(C_1), Beh(C_2))$$

where *Rep* maps the connectivity of the circuits to the equivalent representation (renamings and hidings as in CCS [Mil80]) used in HOP. Informally, PARCOMP *statically computes a closed-form behavioral specification* for an entire module from behavioral specifications of the submodules as well as their interconnections. The algorithm works by statically evaluating programs that use the operator || (our || is closest to CSP's || used in [Hoa84, Chapter-2]). This static evaluation always terminates (for reasons shown later). This static evaluation also capitalizes on the *event hidings* (as in [Mil80]) to achieve a few orders of magnitude speed-up than a naive static evaluation that doesn't. The resultant closed-form behavioral expressions present all producer/consumer relationships very abstracty and readably.

The main reason for achieving this abstraction is now illustrated. Suppose module M is triggered by the controller to produce a value $f(E)$ on output port $!p$ at tick t. Suppose module N is triggered to consume a value x on input port $?p$ (ports $!p$ and $?p$ are connected according to our conventions) and then use x in creating the next data-path state $g(x)$ of N at tick $t+1$. PARCOMP first grinds through such low-level specifications and displays as the net result the following, much more succinct, representation: (i) M produces value $!p$ at tick t; (ii) N attains data-path state $\underline{g(f(E))}$ at tick $t+1$.

On the other hand, if PARCOMP encounters two *non-synchronizing* events, say $e1$ and $e2$, at tick t which are *hidden* (*i.e.* $e1$ and $e2$ are localized within a module), it would report an error, indicating a "synchronization failure". This major side-benefit of PARCOMP therefore helps in detecting sequencing errors ("synchronization failures" in a high-level sense) present in user's stated synchronization requirements, without any simulation. *This should be contrasted with current practices* which involve (for example) running a circuit with test data and *hoping* that this test data would cause the control-sequencer to reveal its sequencing bug, which, in turn, is very indirectly observed (if at all) by seeing the wrong module getting activated, two modules clashing on a bus, etc.

We know of no other comparable work that "deduces behavior from structure" (similar to PARCOMP), other than in very-low level automata theoretic models that introduce one modeling-level state for every *data path* state; our technique introduces one modeling-level state *only* for every control state which are far fewer in number.

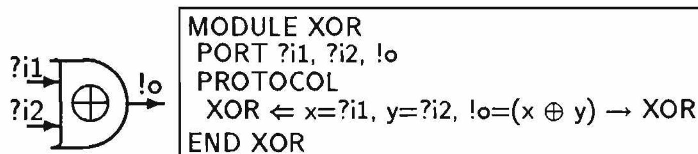

Figure 1: The Absproc Specification of an XOR Gate

We also note that manual controller specification languages such as [AM84,Hen81] could easily be *transliterated* into HOP; in addition if the data paths that these controllers control are also specified in HOP, current VLSI design methodologies could profit from PARCOMP.

3. PARCOMP has been implemented in Common-Lisp. We show some results obtained in using it.

4. The following additional spin-offs from PARCOMP are (briefly) discussed: (i) a new new way of simulating digital circuits (somewhat similar to that in [Mil85]); (ii) formal verification techniques; (iii) A possibility for automating pipelining, using semantics preserving transformations conducted on HOP specifications; (iv) possibility of symbolically simulating VLSI (combinational and sequential) circuits.

The main thrust of this paper will be in presenting HOP and algorithm PARCOMP.

Roadmap

The rest of this paper is organized as follows. Section 2 presents HOP through several examples. Section 3 provides an overview of the semantics of HOP. Section 4 presents PARCOMP through an example. Section 5 outlines work in progress. Though we often indicate where missing details exist in the appendices, *there is no need to read the appendices to follow this paper.* We will present a detailed literature survey in a longer version of this paper.

2 Illustration of HOP

Let us start with two simple modules, an XOR gate and a flip-flop (FF), specify them both in HOP, and be able to read and *intuitively* understand their specifications. These submodules will then be used in an MLS counter, whose behavior we will deduce using PARCOMP.

2.1 An XOR gate

Figure 1 shows salient excerpts from the specification of an XOR gate written in HOP. A specification that specifies a hardware system as a black-box without revealing its internal structure is called an *Absproc* specification. It states enough timing details (external timing protocols) to permit the module to be used in a larger context. It typically includes a large amount of "code" in a purely functional language that specifies how states and values are created.

Though our XOR gate is combinational and is "usable at any time", in a synchronous digital system it gets used only at clock ticks. Here we assume that it is used only during phases a or b of a two-phase clock train.

At each time step, inputs x and y (respectively) are consumed via ports $?i1$ and $?i2$, and a data assertion $(x \oplus y)$ is made on the output port $!o$. No events are declared, and the XOR gate repeatedly performs the above task. The process diagram in figure 2(a) depicts the HOP specification of the

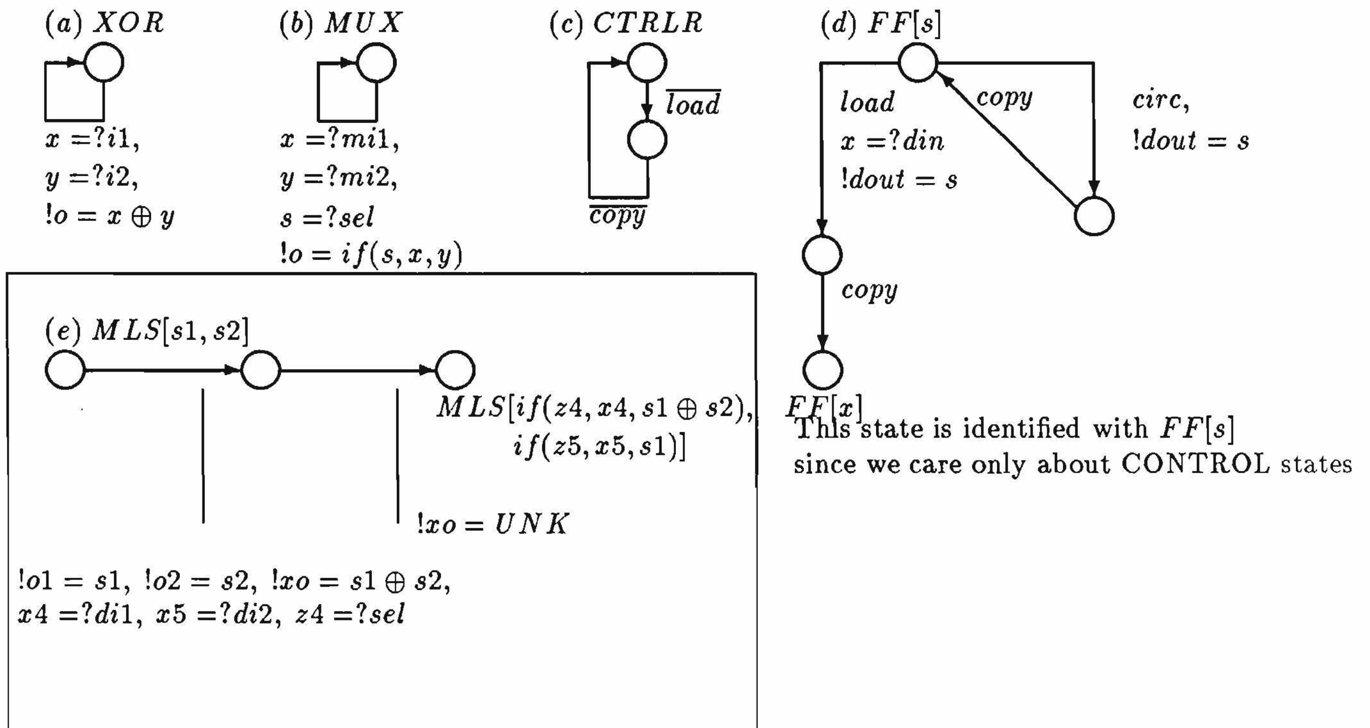

Figure 2: The HOP Processes XOR, MUX, FF, CTRLR and MLS

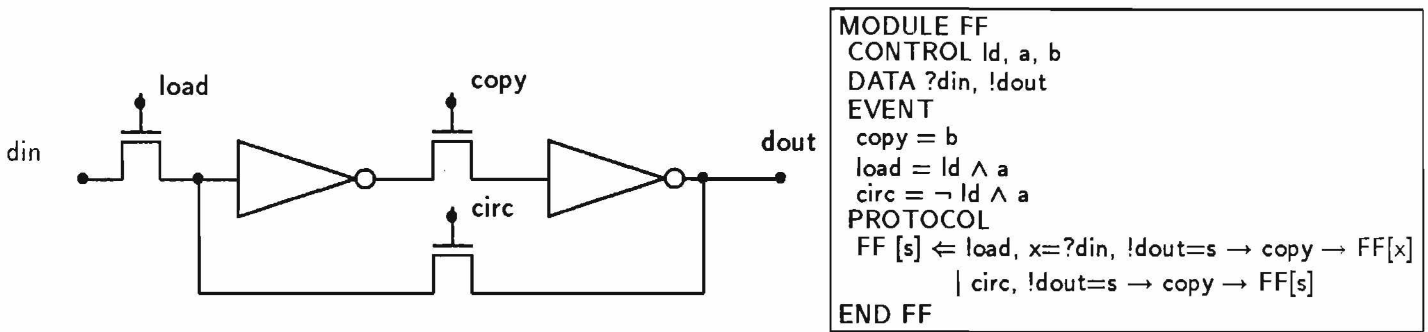

Figure 3: The Circuit and Absproc Spec. of a Two-phase Clocked MS FF

XOR gate. The basic unit of time is the "tick" of one of the phases. *Note:* All our specifications have their "starting phases" as phase-a.

Similarly, a multiplexor can be specified as per figure 2(b), and a CTRLR module (which we will use shortly) in figure 2(c).

2.2 A Two-phase Clocked Master-slave Flip-flop

The circuit schematic of a Flip-flop and its Absproc specification are shown in figure 3. The events to which the Flip-flop *FF* responds to are *load,copy* and *circ*. *load* occurs if during phase *a* the input signal *ld* is asserted; otherwise *circ* occurs. *copy* always occurs during phase *b*. *In figure 3, we do not show the boundary control wires as well as the circuitry to generate the events load, copy and circ.* The starting phase of FF is phase-a.

This is how to read the specification of FF: The CONTROL wires coming into FF are ld, a and b. The DATA wires are ?din, and !dout. The EVENTS copy, load and circ are all input events; we denote an "output event **e**" by $\overline{e}$. Note that events are defined as logic combinations of control wires and CLOCK wires a,b.

Finally, the PROTOCOL section states the following. **The FF process in data-path state s, as represented by FF [s].** (The stuff inside [] is almost always a data path state.) FF[s] offers two "choices" of events: load and circ. These guards are to be mutually exclusive by definition (thereby guaranteeing determinacy). If $\overline{load}$ is asserted by the user of FF while supplying x through ?din, !dout is held at s during phase a, during the next relevant time (→ shows passage of one tick of time) the copy event must occur (phase b ticks). Following this, the behavior is similar to FF[x].

Especially noticeable is the fact that the flip-flop may be read during phase *a* regardless of whether *load* is asserted during phase *a* or not. This fact is always exploited by hardware designers in overlapping "read" and "write" on a flip-flop—but never made explicit in any formal hardware specification language to our knowledge. Many such details, such as "read from cache is OK while write-through is progress in main-memory", can be expressed in HOP.

2.3 A Two Bit Maximal Length Sequence (MLS) Counter

The schematic and Absproc specifications of a two-bit Maximal Length Sequence (MLS) counter are shown in figure 4, and depicted in figure 2(e). **This specification was not manually written, but computed using PARCOMP.** The effect of PARCOMP is to infer the boxed process diagram of the MLS counter from the rest of the process diagrams in figure 2. (Details soon to be presented.)

According to the inferred behavior, initially the *sel* inputs of the MUX modules could be set to permit the loading of the Flip Flops FF1 and FF2 (the data loaded is non-zero). Thereafter,

```
MLS [s1 s2] ⇐
  !o1=s1 !o2=s2 !xo=s1⊕s2 x4=?di1 x5=?di2 z4=?sel z5=?sel → !xo = UNK
  → MLS [ if(z4 x4 s1⊕s2) if(z5 x5 s1) ]
```

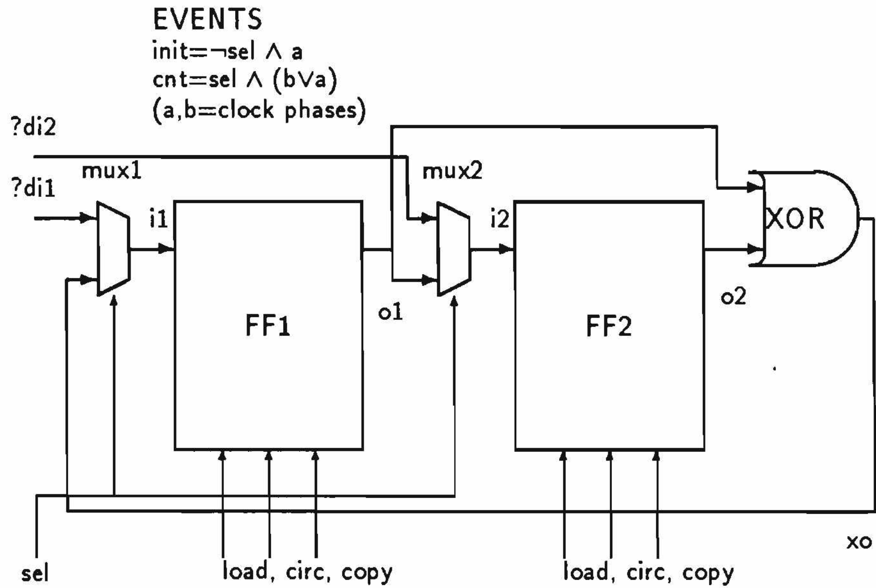

Figure 4: An MLS Counter's Realization

CONNECT
DATANODES
• Rename all submodule ports that meet at a common point to a common "node" name; specify also whether node is hidden; if not hidden, furnish an external port name to this "exported" port.

EVENTNODES
• Rename all submodule events that are to have the same logical value to a common internal event name; specify if hidden; if not hidden, furnish an external event name to this "exported" event.

Figure 5: A Realproc Specification of an MLS Counter

the counter counts through three of its data path states. It is up to the user to wisely use the *sel* input!

2.3.1 Expressing the MLS Counter as a Process Realization

The *Realproc* specification of the MLS counter is given in figure 5. A specification is called a *Realproc* specification if it specifies the ***realization*** of a hardware system as an interconnection of processes. A realproc specification is often easy to write but its behavior is often hard to understand because one has to imagine all possible runs of, and interactions among, the submodules. Simulation helps only in revealing small portions of potential behaviors. Fortunately, PARCOMP deduces all possible behaviors and represents it in a closed form, as an Absproc description.

2.3.2 Lisp Representations of HOP Specifications

To give an overview of the current prototype of the HOP system, we present excerpts from the Absproc specification of the FF module and the Realproc specification of the MLS counter module, in appendix A.6.

3 Overview of the Semantic Basis of HOP and PARCOMP

Here, we deliberately simplify the presentation of the theory behind HOP to leave more room for examples. The abstract syntax of HOP is presented in figure 8, appendix A.1. A HOP specification specifies a collection of processes interacting with each other in lockstep synchrony, at the beats of a (uni or multi-phase) clock. At first glance, a HOP specification can be thought to specify a collection of "communicating Mealy machines". The differences are:

- The events that label the Mealy machine's transitions have a rendezvous [Hoa84] semantics; that is, if machine M_1 would produce an *output* event $\overline{e}$ on its transition TO to be taken at time t and machine M_2 would produce an *input* event e on one of its transitions TI_k to be taken at time t, then the machines can make progress via TO and TI_k; else they *deadlock*. *This deadlock models synchronization failure* because the intended synchronization requirements were not met at time t. This situation is exactly as in [Mil82].
- Transitions are, in general, labeled by *data queries* and *data assertions*. It is via these that data value communications take place. Data assertions define port value bindings that are in effect for one tick. If a data query is meanwhile issued by another module, it gets the value binding defined by the data assertion. Data queries and assertions do not *rendezvous*, thereby giving considerable leeway in process interactions. Nevertheless proper usage of events (that have a rendezvous behavior) is essential for meaningful data transfers. This is one of the major differences between HOP and SCCS. (Note: Data assertions can also model idealised switches by simply asserting an equality constraint.)

The formal operational semantics of HOP is provided in appendix A.2, figures 9 and 10. Basically these figure defines the transition relation $\rightarrow$ via structural induction over the syntax of HOP. **In more plainer terms**, this defines "what all a HOP process can do" for all HOP processes. We illustrate just one of these rules: ***Parcomp***. (Note: We discuss only a simplified form of this rule):

$$\frac{P \xrightarrow{ca1} P',\ Q \xrightarrow{ca2} Q'}{(P \| Q) \xrightarrow{ca1,ca2} (P' \| Q')}$$

According to this definition, if process P can participate in action $ca1$ and transform itself into P', and if process Q can participate in action $ca2$ and transform itself into Q', then $(P \parallel Q)$ can participate in action $ca1, ca2$ and transform itself into $P' \parallel Q'$. The notation $ca1, ca2$ stands for the "action product" of actions $ca1$ and $ca2$ (in the same way action product is defined in SCCS). Basically, the "action product" construct captures the process of interaction among the actions $ca1$ and $ca2$. The concept of action product is a concise way of specifying synchronizations as well as value communications among modules. Figure 9 defines the action product operator.

According to rule ***Parcomp***, computing the parallel composition of two processes P and Q involves:

- Starting of P and Q at their start states S_P and S_Q;
- Clashing all possible actions of P and Q at these states;

- Generating all pairs of next states of P and Q;
- Continuing the above process, *in the presence of value bindings generated from the previous clashings*;
- Stopping when all pairs of reachable control states have been examined.

The procedure is superficially similar to performing a product of two synchronously working automata; pairing the respective states that are *an equal number of transitions away* from the respective start states, and then connecting these pairs of states with transitions labeled by paired events. However there are two key differences.

(i) transitions are labeled by data queries as well as data assertions; their interactions lead to value communications among the processes; (ii) events have to synchronize, or else the process deadlocks.

The termination of PARCOMP is due to the finite number of control state pairs in a cartesian product. We however generate far fewer states due to: (i) the "equal distance merge" that doesn't pair certain states; (ii) pruning synchronization trees [Mil80] by capitalizing on hiding information.

4 Illustration of PARCOMP

4.1 Illustration of PARCOMP on the MLS Counter

- To deduce the Absproc description of the MLS counter, given diagrams 2(a) through (d) and the interconnectivity specification among the submodules (figure 5)—actually a more detailed CONNECT specification as in appendix A.6. We express the goal as:

  MLS [s1,s2] = Connect <Interconnections of the MLS> in Hide {copy,load,i1,i2} in
  (FF1[s1] || FF2[s2] || XOR || MUX1 || MUX2 || CTRLR)
  where FF1 and FF2 are of type FF, and MUX1, MUX2 of type MUX.
  (Hiding p hides ?p and !p; hiding e hides $\overline{e}$ and $\overline{\overline{e}}$; $\overline{\overline{e}}$ is a "synchronized event".)
- The connect specification is handled by renaming all nodes interconnected to common names. Also all variables within processes are renamed to avoid name clashes. Then we have:

  = Hide {?copy,!copy,?load,!load,?i1,!i1,?i2,!i2} in
  (FF1[s1] || FF2[s2] || XOR || MUX1 || MUX2 || CTRLR)
- Observe that the event *circ* is not generated by any of the participant modules. Therefore the choice offered by *FF*1 as well as *FF*2 on *circ* are satisfiable *only* if some module *outside* the boundary of MLS generates *circ*. However, *circ* event is hidden; that is, no module from outside can supply *circ*. Therefore, we immediately prune the arm of FF1 and FF2 labeled by *circ*. In practical terms this means that FF1 and FF2 are always used as shift registers—no recirculation of data is done! (Note: In our actual implementation of the parallel composition procedure, this pruning take a few more fixed-size simple steps.)
- The parallel composition now reduces to: (i) taking the action product of the events offered by all the modules for the first instant of time; (ii) continuing with the remaining instants of time. We write the initial events offered by each of the modules in separate lines below, followed by → which is followed by the collective behavior of the system from the second time instant onwards. One may read-off these lines from the figure 2.

  (Offered by FF1) load, x1=?i1, !o1=s1,
  (Offered by FF2) load, x2=?i2, !o2=s2,
  (Offered by XOR) x3=?o1, y3=?o2, !xo = x3⊕y3,
  (Offered by MUX1) x4=?di1, y4=?xo, z4=?sel, !i1=if(z4,x4,y4),
  (Offered by MUX2) x5=?di2, y5=?o1, z5=?sel, !i2=if(z5,x5,y5),
  (Offered by CTRLR) !load

→
The following behavior under bindings B1 (defined below) :
((!copy → FF1[x1]) || (!copy → FF2[x2]) || XOR || MUX1 || MUX2 || (!circ → CTRLR))

- The binding B1 is generated by the action product occurring during the first time instant. It is:
  {<x1 ← if(z4,x4,s1⊕s2)>, <x2 ← if(z5,x5,s1) >,
  <x3 ← s1>, <y3 ← s2>, <y4 ← s1⊕s2>, <y5 ← s1> }
  The generation of such a binding is required by rule *Data-assns* of figure 10.
- The parallel composition procedure continues by: (i) unraveling the behavior starting at the second time instant; (ii) pruning away all unsynchronized but hidden events (there are none for the MLS); (iii) terminating if all control combinations have been exhausted. In the current example, after expanding the behavior at the second time instant, we would be faced with:
  (FF1[..] || FF2[..] || XOR || MUX1 || MUX2 || CTRLR)
  for some arbitrary arguments to FF1 and FF2. **At this point, our procedure can stop** because it is the control combination <FF1,FF2,XOR,MUX1,MUX2> that we began with. We call this a control combination because it represents an execution control status of all the subprocesses.
- Thus, we get diagram 2(e).

4.2 Results of the Run from Our Implementation

Our implementation took the inputs given in appendix A.6 produced the listing, shown in appendix A.6 (edited for visual clarity). It says that the entry-point is control state (0 0 0 0 0 0). Entering there, we see that the process generates no events, generates a few data assertions that specify ports and expressions (DI and DO). (The strange variable names are obtained after renamings to avoid name clashes.) The next control state is (1 1 0 0 0 1) and the next data path state is also specified. At control state (1 1 0 0 0 1), we perform some actions and come back to control state (0 0 0 0 0 0). This corresponds to figure 2(e). The computation takes a few seconds of elapsed time on a HP-Bobcat running HP Common Lisp.

Roughly ten times more time is taken, on the average, if we do not capitalize on the event hiding information. For another larger example (a FIFO queue), not using the hiding information resulted in the generation/re-examination of roughly a hundred times more control states.

4.3 Statically Detecting Synchronization Failures

We plugged in a bad CTRLR module that has $\overline{copy}$ and $\overline{load}$ switched with respect to figure 2(c). The parallel composition yielded a process with no transitions at all (STOP of [Hoa84]).

Any *finite process*—a process that has a finite sequence of actions leading to a deadlock—is useless for building digital systems. A module, after all, has to run forever! Upon detecting a deadlock our system reports the cause for it; in this case, revealing that the user made a high-level sequencing error, thereby preventing the event *load* from getting synchronized. *No simulation was performed* to reveal this synchronization error.

We can similarly detect this, and other errors such as: (i) errors due to omission of data port interconnections, causing UNKs to propagate; (ii) two syntactically different values clashing on a port, for the MLS, and much larger examples.

5 Work in Progress

This section touches on many design automation issues centered around HOP. Due to the page limits, we present only one in detail. The rest are in appendix A.5.

```
TESTER ⇐ !sel=1 !dii=0 !di2=1 → COUNT[4]
COUNT[N] ⇐ if (N=0) then PRINT_N_STOP
                  else !sel=0 → COUNT[N-1]
PRINT_N_STOP ⇐ print("Counting test over") → STOP
```

Figure 6: A Tester Process for an MLS Counter, Expressed in HOP

```
(MLS[s1,s2] || TESTER) ⇐
 ( !sel=1, !di1=0, !di2=1, !o1=s1, !o2=s2, !xo=s1⊕s2 →
  ( !xo=UNK, MLS[if(z4,x4,s1⊕s2), if(z5,x5,s1)] ) || COUNT[4] )
  under bindings
{< x4 ← 0 >, < x5 ← 1 >, < z4 ← 1 >, < z5 ← 1 >}
```

If we start FF1 and FF2 in states 0 and 1 respectively, and unravel the parallel composition via the simulator, we can obtain the following:

```
( (!o1=0, !o2=1, !xo=1 → <some outputs> → COUNT[2]) || MLS[1,0])
   and then
( (!o1=1, !o2=0, !xo=1 → <some outputs> → COUNT[1]) || MLS[1,1])
   and then
( (!o1=1, !o2=1, !xo=0 → <some outputs> → COUNT[0]) || MLS[0,1])
   thus completing one cycle.
```

As will be seen shortly, this is also the basic technique for symbolic simulation.

Figure 7: Parallel Composition of a MLS Counter and Its Tester

A New Approach to Simulation

A great advantage of HOP is that we do not need a separate simulation command language; HOP itself suffices! Example: the tester process defined in HOP to test the MLS counter figure 6. This tester process sets the !sel port to 1 (to permit loading the flip-flops), !di1 to 0 and !di2 to 1 (the data items to be loaded), all during the first time instant. Starting at the second time instant, the behavior of TESTER is the same as that of COUNT[4]. The COUNT[N] process counts N times in a loop. Until N gets to zero, the !sel line is kept at 0, thereby permitting data to flow among the submodules. When the count runs out, a message is printed (an event only for humans) and the simulation halts. Halting is simply achieved by virtue of the fact that STOP is a process that is capable of no actions at all.

In order to simulate a circuit: (i) statically compute the parallel composition of the module to be tested with a "tester" process; some errors could surface now; if not we get (we believe) *faster-to-simulate functional expressions*; (ii) simply "unravel" the result of the parallel composition. The result of taking such a parallel composition is given in figure 7.

Pipelining

Some languages (e.g. SLIM [Hen81]) have the ability to specify automata, and additionally state that certain outputs be generated *earlier* or *later* relative to other events. These help in pipelining systems, albeit *manually*. In HOP, we can model the idea of pipelining used in SLIM. The idea is to substitute one controller for another in a subsystem and check that the overall behavior is unaffected, except for the gained speed. We have some examples that support our research in this direction. One example is given in appendix A.8.

Formal Verification and Symbolic Simulation

We have currently completed the formal verification of a large ASIC that we are designing. We could present excerpts from this in the final paper. Some details of the latter are in appendix A.5.1.

References

[AM84] P. Agrawal and M. J. Meyer. Automation in the Design of Finite-State Machines. *VLSI design*, 5(9), September 1984.

[Bar81] Mario R. Barbacci. Instruction Set Processor Specifications (ISPS): The Notation and Its Applications. *IEEE Transactions on Computers*, C-30(1):24–40, January 1981.

[Cam84] Raul Camposano. Automatic Datapath synthesis from DSL specifications. In *Proceedings International Conference on Computer Design: VLSI in Computers*, pages 630–635, 1984.

[CGM86] Albert Camilleri, Michael C. Gordon, and Tom Melham. Hardware Specification and Verification using Higher Order Logic. In *Proceedings of the International Working conference From HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble (IFIP)*, North-Holland, 1986.

[FTG88] Richard Fujimoto, Jya-Jang Tsai, and Ganesh Gopalakrishnan. The Roll Back Chip: Hardware Support for Distributed Simulation Using Time Warp. In *(Accepted for publication in) The Distributed Simulation Conference, San Diego, CA*, 1988.

[GFK87] Ganesh Gopalakrishnan, Richard Fujimoto, and Jed Krohnfeldt. A Language for Specifying Lockstep-synchronous Deterministic Processes: New Communication Mechanisms and Applications to Hardware Design. In *(Submitted to) ACM Principles of Programming Languages, 1988*, 1987.

[Gop86] Ganesh C. Gopalakrishnan. *From Algebraic Specifications to Correct VLSI Systems.* PhD thesis, State University of New York, December 1986.

[GS87] Ganesh C. Gopalakrishnan and Mandayam K. Srivas. *Implementing Functional Programs Using Mutable Abstract Data Types.* Technical Report, Dept of Computer Science, Univ. of Utah, Salt Lake City, UT 84112, May 1987. (Also submitted to Information Processing Letters.).

[GSS86] Ganesh C. Gopalakrishnan, Mandayam K. Srivas, and David R. Smith. From Algebraic Specifications to Correct VLSI Circuits. In *Proceedings of the International Working conference From HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble (IFIP)*, North-Holland, 1986.

[Hen81] John Hennesey. SLIM: A Simulation and Implementation Language for VLSI Microcode. *Lambda*, 1981. Second Quarter.

[Hoa84] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1984.

[Joh84] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* The MIT Press, 1984. An ACM Distinguished Dissertation-1983.

[MC80] C. A. Mead and L. Conway. *An Introduction to VLSI Systems.* Addison Wesley, 1980.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980. LNCS 92.

[Mil82] Robin Milner. *Calculii for Synchrony and Asynchrony.* Technical Report CSR-104-82, Univ. of Edinburg, 1982. Internal Report.

[Mil85] 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.

[She85] Mary Sheeran. muFP... In *Proceedings of the Functional Programming and Computer Architecture Conference*, Springer-Verlag, LNCS 201, September 1985. Nancy, France.

Notes: The abstract syntax has to be augmented with the following rules:

- A choice (|) may not be offered among different output events. Essentially, being in a control state determines the output events generated uniquely.
- Choices must contain mutually exclusive input events in their input guards (*initials* citeCSP-book). In reality, the input guards are also exhaustive: all other events lead the process to *STOP*, i.e. deadlock.

process_definition ::= *process_id* `[` *parameters* `]` = *rename_process*
rename_process ::= *rename_function hide_process*
    | *hide_process*
    | `if` (*condition, rename_process, rename_process*)
rename_function ::= `rename` *event_port_names* `to` *event_port_names* `in`
hide_process ::= *hide_function par_process* | *par_process*
hide_function ::= `hide` *event_port_names* `in`
par_process ::= *deterministic_choice*
    | *deterministic_choice* `||` *par_process*
    | *process_id* `[` *actuals* `]`
    | *process_id* `[` *actuals* `]` `||` *par_process*
    | *rename_process*
    | *par_process* `||` *rename_process*
deterministic_choice ::= `(` *rest_of_choices* `)`
rest_of_choices ::= *seq_process*
    | *seq_process* `|` *rest_of_choices*
seq_process ::= *process_id*
    | *process_id* `[` *actuals* `]`
    | *compound_event* `->` *seq_process*
compound_event ::= *simple_event*
    | *simple_event* `,` *compound_event*
    | *dataio_assertion*
    | *dataio_assertion* `,` *compound_event*
simple_event ::= *eventid* | $\overline{eventid}$ | $\overline{\overline{eventid}}$ | $\overline{idle}$
dataio_assertion ::= *variable* = ?*portid* | !*portid* = *expression*

Figure 8: Abstract Syntax of HOP, with terminals in `teletype` font

A Appendix

A.1 The Abstract Syntax of HOP

See figure 8.

A.2 The Formal Operational Semantics of HOP

See figures 9 and 10. Some explanations are now presented.

Separately defined processes with different event and port names may be made to interact with each other by renaming their interface events and ports to common names. A processes may be prevented from interacting with another through an event or a data port by hiding that event or data port.

We have found that having three kinds of events *input*, *output* and *synchronized* is helpful both semantically and pragmatically in the following ways:

- Specifications are made to relate to practical reality (driven signals .vs. input signals) better. In addition, our definition of event products (figure 9) is based on our intuition of how synchronous hardware systems work. In order to capture this intuition faithfully, we have found it necessary to have these three kinds of events.
- to have broadcast semantics for events ($\overline{\overline{e}}$ behaves like $\overline{e}$ as far as event product goes; both of them can "meet and annihilate" an input event.

$$
\begin{aligned}
e,\, e &\Rightarrow e && (1)\\
e,\, \overline{e} &\Rightarrow \overline{\overline{e}} && (2)\\
e,\, \overline{\overline{e}} &\Rightarrow \overline{\overline{e}} && (3)\\
\overline{e},\, \overline{e} &\Rightarrow \overline{e} && (4)\\
\overline{\overline{e}},\, \overline{e} &\Rightarrow \overline{\overline{e}} && (5)\\
\overline{\overline{e}},\, \overline{\overline{e}} &\Rightarrow \overline{\overline{e}} && (6)\\
port = E,\ x = port &\Rightarrow port = E,\ \textit{with binding}\ [E/x] && (7)
\end{aligned}
$$

Figure 9: Definition of *Action Product* in HOP

- to distinguish a synchronized action ($\overline{\overline{e}}$) from an unsynchronized action ($\overline{e}$) with respect to hiding; ***Hiding an unsynchronized event causes deadlock whereas hiding a synchronized event is equivalent to replacing that event with an idling event.***
- to define a function ***interpret*** that takes a process and reveals its execution history, by traversing that path (unique due to determinacy) of the synchronization tree [Mil80] that is labeled solely by synchronized events and data assertions. The third precondition of rule for parallel composition as well as the notes given in figure 8 guarantee the determinacy of HOP.

In figure 9 we provide the action product axioms of HOP. They define the way in which we wish to see simultaneously occurring events and data assertions interact. Specifically: synchronization is defined; unsynchronized simultaneous events are allowed, hoping that under future parallel compositions these events would get synchronized. The relation $\Rightarrow$ may be read as "may be simplified to". The action product operator ',' is commutative and associative with identity element $\overline{idle}$.

In figure 10, we present the operational semantics of HOP as a labeled transition system. For each compound action ca, the relation $\xrightarrow{ca}$ is the smallest relation closed under the rules given in figure 10. These rules are to be applied only after a canonical representation for action products defined by figure 9 has been obtained.

[*Action* and *Det-choice*] These are basis cases of the transition relation, with *Action* being a special case of *Det-choice*. The rule *Det-choice* says that a process defined as a deterministic choice of compound actions ca_i can evolve to P_i via ca_i.

[*Parcomp*] This rule defines the parallel composition of two processes. Consider the first two preconditions of this rule (for ease of understanding). If P and Q can evolve via $ca1$ and $ca2$, $P \parallel Q$ can evolve via $ca1, ca2$, the action product of $ca1$ and $ca2$. The third precondition of this rule is what guarantees determinacy; if Q can disrupt the mutually exclusive nature of the choices offered by P, the parallel composition of two processes P and Q doesn't have any transitions defined.

[*Data-assns*] This rule captures how individual data assertions interact upon parallel composition and impart a binding to variables present in the receiving process. The cases of contradicting assertions (two modules driving a data bus with different values) is handled naturally; a port identifier would be equated to two different values. By admitting a value HIZ (to model open busses) and suitable resolutions of bindings such as $port = HIZ, port = 22 \Rightarrow port = 22$ "wired functions" also can be modeled. Likewise modeling bidirectional flow of information, such as in a pass transistor, is easy; both ends of the pass transistor are asserted to have the same value.

[*Hiding-sync*] When event e is hidden from process P, all synchronized events $\overline{\overline{e}}$ in P are replaced with $\overline{idle}$, the identity event for action products.

[*Hiding-unsync*] When an unsynchronized event $ca1$ is hidden from P, the branch of the synchronization tree of P labeled by $ca1$ (as well as the subtree following $ca1$) is pruned.

[*Hiding-dout*, *Hiding-din* and *Renaming*] Obvious.

[*Conditionals* and *Recursion*] The rule for recursion is essentially that defined on [Mil82, page 8].

Action $(ca \rightarrow P) \xrightarrow{ca} P$

Det-choice $(|_i \, ca_i \rightarrow P_i) \xrightarrow{ca_i} P_i$

Parcomp $\dfrac{P \xrightarrow{ca1} P',\ Q \xrightarrow{ca2} Q',\ (P \xrightarrow{e1,ca1} P',\ P \xrightarrow{e2,ca2} P'',\ e3 \in \{\overline{e1}, \overline{\overline{e1}}\},\ e4 \in \{\overline{e2}, \overline{\overline{e2}}\} \Rightarrow not(Q \xrightarrow{e3,e4,ca3} Q'))}{(P \| Q) \xrightarrow{ca1,ca2} (P' \| Q')}$

Data-assns $\dfrac{P \xrightarrow{(x=p),ca1} P',\ Q \xrightarrow{(p=E),ca2} Q'}{(P \parallel Q) \xrightarrow{(p=E),ca1,ca2} (P' \parallel Q')\,[E/x]}$

Hiding-sync $\dfrac{P \xrightarrow{ca} P'}{\texttt{Hide}\ e\ \texttt{in}\ P \xrightarrow{ca[\overline{idle}/\overline{e}]} \texttt{Hide}\ e\ \texttt{in}\ P'}$

Hiding-unsync $\dfrac{P \xrightarrow{ca1} P',\ P \xrightarrow{ca2} P'',\ e\ or\ \overline{e} \in ca1}{(\texttt{Hide}\ e\ \texttt{in}\ P) \xrightarrow{ca2} (\texttt{Hide}\ e\ \texttt{in} P'')}$

Hiding-dout $\dfrac{P \xrightarrow{ca,p=E} P'}{\texttt{Hide}\ p\ \texttt{in}\ P \xrightarrow{ca} \texttt{Hide}\ p\ \texttt{in}\ P'}$

Hiding-din $\dfrac{P \xrightarrow{ca,x=p} P'}{\texttt{Hide}\ p\ \texttt{in}\ P \xrightarrow{ca} \texttt{Hide}\ p\ \texttt{in}\ P'\, with\ x\ unbound}$

Renaming-e $\dfrac{P \xrightarrow{e} P'}{\texttt{Rename}\ e\ \texttt{to}\ e1\ \texttt{in}\ P \xrightarrow{e1} \texttt{Rename}\ e\ \texttt{to}\ e1\ \texttt{in} P'}$

Renaming-$\overline{e}$ $\dfrac{P \xrightarrow{\overline{e}} P'}{\texttt{Rename}\ e\ \texttt{to}\ e1\ \texttt{in}\ P \xrightarrow{\overline{e1}} \texttt{Rename}\ e\ \texttt{to}\ e1\ \texttt{in} P'}$

Renaming-port $\dfrac{P \xrightarrow{da} P',\ da\ uses\ p}{\texttt{Rename}\ p\ \texttt{to}\ p1\ \texttt{in}\ P \xrightarrow{da[p1/p]} \texttt{Rename}\ p\ \texttt{to}\ p1\ \texttt{in} P'}$

Conditional

$$\dfrac{P1 \xrightarrow{ca} P'}{(\texttt{if}\ true\ \texttt{then}\ P1\ \texttt{else}\ P2) \xrightarrow{ca} P'}$$

$$\dfrac{P2 \xrightarrow{ca} P'}{(\texttt{if}\ false\ \texttt{then}\ P1\ \texttt{else}\ P2) \xrightarrow{ca} P'}$$

Recursion $\dfrac{P_i[fix\ \overline{X}.\overline{P}/\overline{X}] \xrightarrow{ca} P'}{fix_i\ \overline{X}.\overline{P} \xrightarrow{ca} P'}$

Notes: The transition for *Parcomp* may be strengthened to include: (i) clashing outputs on a bus; (ii) two modules generating an output event (seldom happens, and skews make this dangerous!).

Figure 10: Operational Semantics of HOP

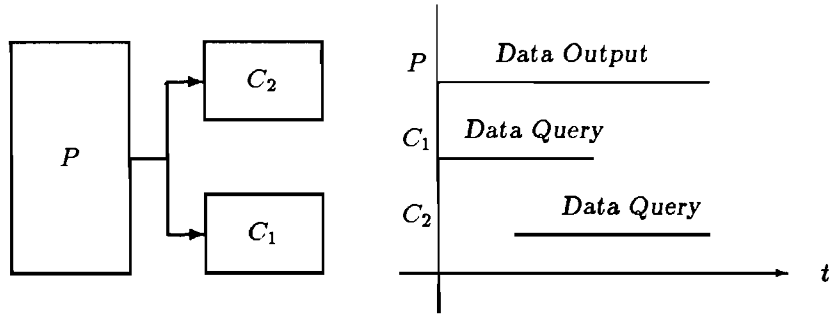

Figure 11: Use of Data Assertions

A.3 The Computational Model Underlying HOP

A collection of processes P_1, P_2, ..., P_N, share a global clock and interact with each other on each clock beat through a finite set of named *events* e_i, $\overline{e_j}$, etc. At each time step, a process P: waits for one of several mutually exclusive events $e_1,...,e_m$ to come true (offers a "deterministic choice" in the sense of cite[Chapter-2]Hoare-book) or idles. A process that idles has no effect on the other processes. A process P offering a choice $\mathcal{E} = e_1, ..., e_m$ at time step t deadlocks if none of $\overline{\mathcal{E}} = \overline{e_1}, ..., \overline{e_m}$ are produced by some other process during time step t. If, however, $\overline{e_j} \in \overline{\mathcal{E}}$ is produced, P synchronizes on e_j and continues to behave like some process, P_j. Each output event $\overline{e_j}$ can synchronize with more than one input event e_j. Non-synchronizing events e_1,$\overline{e_2}$, etc. may occur simultaneously, too.

So far our computational model has resembled that of SCCS citeMilner-sccs except for determinacy and broadcast semantics for output events. But now we start differing. Each process possesses a finite set of named *data input* and *data output* ports p_i. A set of ports can be regarded to be connected if they have the same name. Such a connection is called a *node*. At each time step a process can make *data assertions* on its output ports or *query data* from its input ports. A data assertion is written $!p = E$ where $!p$ is a port (! to indicate it is an output port) and E is an expression denoting some value (such as integers). A data query is written $x = ?p$ where x is a variable that has not been used so far in the lexical scope of the process description (? to indicate input direction). Data assertions and data queries on a set of connected ports are meant to interact; therefore if data assertion $!p = E$ is true at time t, a data query $x = ?p$ binds x to E in the process making the query. However, data assertions may be made without any contemporaneous queries going on; or queries may be properly contained in the interval in which assertions are made. In this sense, value communications (as opposed to synchronization signals) between processes is *not* through rendezvous, unlike all existing Calculii of Communicating Systems.

Also note that the synchronization behavior of a process is solely determined by events. A proper synchronization skeleton of events is vital for meaningful data communications. Yet there is considerable flexibility in the manner in which data assertions can meaningfully interact (see figure 11). We could not find any way to satisfactorily model such situations ***modularly*** in existing Calculii of Communicating Systems ("CCSes") or other process specification languages. In existing CCSes a process performing an output action cannot make progress without another process performing a matching input. Also, using existing CCSes we had to explicitly wire in timings into process descriptions. Despite this extension, we have a simple and elegant operational semantics for HOP A.2, much like that for SCCS citeMilner-sccs. HOP also provides *conditional expressions* of the form $P[x] \Leftarrow$ ***if*** $p(x)$ ***then*** $P[f(x)]$ ***else*** $Q[x]$. This then is HOP's underlying computational model.

A.4 The Parallel Composition Algorithm

Input: An expression **Hide** HS *in* $\parallel \{P_i[\overline{X_i}], ..., C_j[\overline{X_j}], ...\}$ for $i \in \{1..m\}$, $j \in \{1..n\}$. C_j are conditional processes of the form $C_j[\overline{X_j}] =$ **if** q_j **then** $T_j[g_j(\overline{X_j})]$**else** $F_j[h_j(\overline{X_j})]$ and P_i are non-conditional processes of the form $P_i[\overline{X_i}] = y_i \; : \; initials_i \rightarrow R_i(y_i)$; Each P_i offers a set of initial

choices $initials_i$ and for each choice y_i that is offered, the future behavior of P_i is $R_i(y_i)$. HS is the *Hidden Set*, the set of events and ports hidden from the parallel composition.

Output: A behaviorally identical process $P[\overline{X_i}, ..., \overline{X_j}, ...]$.

Method: A *done-list* is maintained for each parallel composition $\| \{P_i[\overline{X_i}], ...\}$ that has already been computed. Upon getting a call for performing parallel composition, the *done-list* is first consulted.

- If the requested parallel composition is in the *done-list*, return. Else enter it in the *done-list* and proceed as follows.
- Combine all conditional processes into one conditional process C. Combining two conditional processes is done as follows:

$$C_1[\overline{X_1}] = \texttt{if } q_1 \texttt{ then } T_1[g_1(\overline{X_1})] \texttt{ else } F_1[h_1(\overline{X_1})]$$

$$C_2[\overline{X_2}] = \texttt{if } q_2 \texttt{ then } T_2[g_2(\overline{X_2})] \texttt{ else } F_2[h_2(\overline{X_2})]$$

$$\begin{aligned} C_1[\overline{X_1}] \,\|\, C_2[\overline{X_2}] \;=\; & \texttt{if } (q_1 \wedge q_2) \texttt{ then } T_1[g_1(\overline{X_1})] \,\|\, T_2[g_2(\overline{X_2})] \\ & \texttt{else if } (q_1 \wedge not(q_2)) \texttt{ then } T_1[g_1(\overline{X_1})] \,\|\, F_2[h_2(\overline{X_2})] \\ & else\ ...etc.\ (all\ four\ combinations) \end{aligned}$$

- Now we are left with the task of computing **Hide** HS *in* $\| \{P_i[\overline{X_i}], ..., C\}$. Let C be of the form

$$\texttt{if } q_1 \texttt{ then } C_1[g_1(\overline{X_1})] \texttt{else if } q_2 \texttt{ then } C_2[g_2(\overline{X_2})] etc.$$

  $\| \{P_i[\overline{X_i}], ..., C\}$ reduces to a conditional process with q_i as the conditions. This conditional has in it parallel compositions of the form $\| \{P_i[\overline{X_i}], ..., C_i\}$. that is (recursively) computed. Eventually we are faced with composing non-conditional processes in parallel. We take this up next.
- Consider $\| \{P_i[\overline{X_i}], ...\}$. Let each P_i be

$$\begin{aligned} P_i[\overline{X_i}] \;=\; & ca_i^1 \rightarrow R_i^1[f_i^1(\overline{X_i})] \\ | \;& ca_i^2 \rightarrow R_i^2[f_i^2(\overline{X_i})] \\ | \;& ... \\ | \;& ca_i^{n_i} \rightarrow R_i^{n_i}[f_i^{n_i}(\overline{X_i})] \end{aligned}$$

- Generate tuples

$$T = < ca_1^{x_1},\ ca_2^{x_2},\ ...ca_m^{x_m} >$$

  i.e. a tuple of the x_1th initial compound action offered by P_1, the x_2th initial compound action offered by P_2, etc. This tuple T is assumed to be the irreducible form arrived at after applying the action product rules of figure 9. According to the rule for parallel composition (*Parcomp* of figure 10), all such tuples would become the initial choices of the resultant process. Following such choices, the resultant process would continue to behave like $\| \{R_1^{x_1}[f_1^{x_1}(\overline{X_1})] R_2^{x_2}[f_2^{x_2}(\overline{X_2})], ...\}$. However using the hiding information HS, we can prune many of these choices. In particular,

  - those tuples T that contain *unsynchronized* events e or $\overline{e}$ that belong to HS are dropped, and the corresponding arm of the synchronization tree is pruned;
  - those tuples T that contain *synchronized* events $\overline{\overline{e}}$ that belong to HS are replaced by $T[\overline{idle}/\overline{\overline{e}}]$.

- In computing

$$\| \{R_1^{x_1}[f_1^{x_1}(\overline{X_1})], R_2^{x_2}[f_2^{x_2}(\overline{X_2})], ...\},$$

  the bindings generated by taking action products of the members of T are taken into account. Specifically, we construct a *let* block containing these bindings. □

A.5 Some Details of Work in Progress

A.5.1 Symbolic Simulation

Symbolic simulation requires us to carry around non-ground (with free variables) expressions in "unravelling" the behavior of the result of parallel composition. Handling non-ground functional expressions during symbolic simulation has the following problems:

- Simplification of the functional expressions that arise during simulation is not an easy problem. In general it involves reasoning in various mathematical theories. Fortunately, we observe (from the examples in [Gop86]) that many such theories are simple and therefore reasoning in them can be automated.
- More challenging is the problem of deciding the path to be followed upon encountering a conditional branch (the *"if"* process in non-terminal *rename_process*, figure 8).

At present we have the following ideas: when a conditional expression (such as (N=0) in the specification of process COUNT) is encountered, the user is queried for a suitable decision. The user's response (say "true") is recorded and the simulation proceeds along the prescribed direction. It is then the responsibility of the simulator to make sure that future evaluations (including future queries from the user) are checked against past responses of the user for consistency. A mixed approach of actual and symbolic simulation is also a possible solution.

A.6 Lisp Representations of HOP Specifications

```
EXCERPTS FROM THE ABSPROC SPECIFICATION OF FF
---------------------------------------------
((name .   ff)
 (kind   . absproc)
 ...
 (port   .      ((din   . (I . BIT))     ; port name, direction, type
                 (dout  . (O . BIT)) ))
 ...
 (event  .      ((copy  .  I)            ; event name and direction
                 (load  .  I)
                 (circ  .  I)))
 (protocol .    ( (ff   . ( (param . ((s . state) ) )
                            (iovar . ( (x . BIT)  ) )
                            (body   .
  ;----------------------------------------
  (CHOICE
    (SEQ  (SIMULT (I load) (DI x din) (DO s dout))
          (SIMULT (I copy))
          (BECOME ff (x)))
    (SEQ  (SIMULT (I circ) (DO s dout))
          (SIMULT (I copy))
          (BECOME ff (s)))
  ))))))
  (defun    .   nil    )                 ; no functions are defined here.
)                           ; if any are needed, supply name, args and
                            ; a lambda expression.
```

```
EXCERPTS FROM THE REALPROC SPECIFICATION OF THE MLS COUNTER
-----------------------------------------------------------
((name .   mls)
 (kind   . realproc)      ; param,const,type sections are also usable in an Absproc spec.
 (param .         nil )   ; Size parameters (e.g. wordlength) can be specified
 (const  .        nil )   ; constants as in Pascal
 (type   .        nil )   ; type definitions as in Pascal

 (port   .        ((cdi1  . (I . BIT))  ; port name, direction, type
```

```
                (cdi2  . (I . BIT))
                (csel  . (I . BIT))
                (co1   .  (O . BIT))
                (co2   .  (O . BIT))))
  (event   .       nil  )
  (subprocess . (  (FF1 .    (ff   )) ; FF1 and FF2 are instances of ff.
                   (FF2 .    (ff   )) ; one may supply actual size
                   (MUX1 . (mux  )) ; parameter also when instantiating
                   (MUX2 . (mux  )) ; these submodules. e.g. wordlength = 16.
                   (XOR  . (xor  )) ; HOP specs. can thus have a nice
                   (ctrlr . (ctrlr)))) ; tailorable structural hierarchy.

  (protocol .  ( (mls .
   ...
  (CONNECT  ; Specify the following for each data node and event node:
            ; Internal name, external name, hidden or not,
            ; list of submodules and their ports that meet at this node.

    (datanode . (
              (nmi1        cdi1  nil  ( (MUX1 . mi1) )
              )
              (nmi1        cdi2  nil  ( (MUX2 . mi1) )
              )
              (nsel        csel  nil  ( (MUX1 . sel) (MUX2 . sel) )
              )
              (no1         co1   nil  (  (FF1 . dout) (MUX2 . mi2)
                                         (XOR . i1  ) )
              )
              (no2         co2   nil  (  (FF2 . dout) (XOR . i2 ) )
              )
              (nmo1        UNK-PORT hidden ( (MUX1 . mo) (FF1 . din) )
              )
              (nmo2        UNK-PORT hidden ( (MUX2 . mo) (FF2 . din) )
              )
              (nxo         UNK-PORT hidden ( (XOR . xo) (MUX1 . mi2) )
              )))
   (eventnode . ( (nload   UNK-EVENT  hidden ( (CTRLR . load) (FF1 . load)
                                               (FF2 . load) )
                )
                (ncopy   UNK-EVENT  hidden ( (CTRLR . copy) (FF1 . copy)
                                             (FF2 . copy) )
                )
                (ncirc1  UNK-EVENT  hidden ( (FF1 . circ) ))
                (ncirc2  UNK-EVENT  hidden ( (FF2 . circ) ))
              )
   ))))))
```

THE RESULT OF PARALLEL COMPOSITION YIELDING THE MLS PROCESS

Explanation:

Processes are represented as a structure with the indicated fields (ENVIRONMENT, etc.).
ENVIRONMENT, EVENTS, DATA-ASSNS, NEXT-CTRL-STATE and NEXT-DP-STATE
are ''parallel lists'', i.e. their ith entries correspond to each other for all i.

A process may be viewed as a function from
events to processes (as in [Hoa84].) Such functions can
be conveniently specified using arrays. This is what we do.
At each array index, specify the ''current state'' actions; then
specify the next control state (another index into the array) and

the next data path state. The data path state can be modified
only during process calls (the ''BECOME'' construct in the Lisp
representation).

Let index (0 0 0 0 0 0) (control state) correspond to time t.
We now study the following listing.

At time t the ENVIRONMENT specifies the bindings in effect
for the data-path state variables as well
as the I/O variables. There are no events on which the MLS will
rendezvous; it will simply scoop up all the data assertions present
at that time and march ahead. There are some data assertions made
at time t. At time t+1, we enter control state (1 1 0 0 0 1).
The data path state remains the same (FF1 and FF2 are essentially
at ''s''). At time t+1,... read on against (1 1 0 0 0 1). We come
back to control state (0 0 0 0 0 0) after that.

The NEXT-DP-STATE has some NILs indicating that XOR, MUX1 and MUX2
don't have data path states.

```
ENTRY CTRL STATE            : (0 0 0 0 0 0)
-------------------------------------------
PROCESS MLS IS :
----------------

(0 0 0 0 0 0):
     ENVIRONMENT    ((XOR$XOR$Y   . FF2$FF$S)
                     (MUX1$MUX$MY . (XOR-FN XOR$XOR$X XOR$XOR$Y))
                     (FF1$FF$X    . (MUX-FN MUX1$MUX$MX MUX1$MUX$MY MUX1$MUX$MS))
                     (MUX2$MUX$MY . FF1$FF$S)
                     (XOR$XOR$X   . FF1$FF$S)
                     (FF2$FF$X    . (MUX-FN MUX2$MUX$MX MUX2$MUX$MY MUX2$MUX$MS)))
     EVENTS          nil
     DATA-ASSNS     (((DO (MUX-FN MUX2$MUX$MX MUX2$MUX$MY MUX2$MUX$MS) NMO2)
                       (DI MUX2$MUX$MS CSEL)
                       (DO FF1$FF$S CO1)
                       (DI MUX2$MUX$MX CDI2)
                       (DO (MUX-FN MUX1$MUX$MX MUX1$MUX$MY MUX1$MUX$MS) NMO1)
                       (DI MUX1$MUX$MS CSEL)
                       (DO (XOR-FN XOR$XOR$X XOR$XOR$Y) NXO)
                       (DI MUX1$MUX$MX CDI1)
                       (DO FF2$FF$S CO2)))
     NEXT-CTRL-STATE ((1 1 0 0 0 1))
     NEXT-DP-STATE (((FF1$FF$S) (FF2$FF$S) NIL NIL NIL NIL)))

(1 1 0 0 0 1):
     ENVIRONMENT    ((FF2$FF$X    . (MUX-FN MUX2$MUX$MX MUX2$MUX$MY MUX2$MUX$MS))
                     (XOR$XOR$X   . FF1$FF$S)
                     (MUX2$MUX$MY . FF1$FF$S)
                     (FF1$FF$X    . (MUX-FN MUX1$MUX$MX MUX1$MUX$MY MUX1$MUX$MS))
                     (XOR$XOR$Y   . FF2$FF$S)
                     (MUX1$MUX$MY . (XOR-FN XOR$XOR$X XOR$XOR$Y)))
     EVENTS          nil
                    ((SIMULT))
     DATA-ASSNS     (((DI MUX1$MUX$MS CSEL)
                       (DO (MUX-FN MUX1$MUX$MX MUX1$MUX$MY MUX1$MUX$MS) NMO1)
                       (DI MUX2$MUX$MX CDI2)
                       (DI MUX2$MUX$MY CO1)
                       (DI MUX2$MUX$MS CSEL)
                       (DO (MUX-FN MUX2$MUX$MX MUX2$MUX$MY MUX2$MUX$MS) NMO2)
```

$$
\begin{aligned}
CTR[s] &= load,\ x = cdi,\ cdo = s \rightarrow CTR[x] \\
&\mid up,\ cdo = s \rightarrow CTR[up(s)] \\
&\mid down,\ cdo = s \rightarrow CTR[down(s)] \\
&\mid cdef,\ cdo = s \rightarrow CTR[s] \\
MEM[s] &= read,\ a = cdo \rightarrow MEM1[s,a] \\
&\mid write,\ a = cdo,\ d = mdi \rightarrow MEM[write(s,a,d)] \\
&\mid mdef \rightarrow MEM[s] \\
MEM1[s,a] &= read,\ mdo = read(s,a),\ a = addr \rightarrow MEM1[s,a] \\
&\mid write,\ mdo = read(s,a),\ a = cdo,\ d = mdi \\
&\quad \rightarrow MEM[write(s,a,d)] \\
&\mid mdef,\ mdo = read(s,a) \rightarrow MEM[s] \\
SCTL &= reset,\ \overline{mdef},\ \overline{cdef} \rightarrow \overline{load},\ \overline{mdef} \rightarrow SCTL \\
&\mid push,\ \overline{mdef},\ \overline{cdef} \rightarrow \overline{up},\ \overline{mdef} \rightarrow \overline{cdef},\ \overline{write} \rightarrow SCTL \\
&\mid pop,\ \overline{mdef},\ \overline{cdef} \rightarrow \overline{down},\ \overline{mdef} \rightarrow SCTL \\
&\mid top,\ \overline{mdef},\ \overline{cdef} \rightarrow \overline{cdef},\ \overline{read} \rightarrow SCTL \\
&\mid sdef,\ \overline{mdef},\ \overline{cdef} \rightarrow SCTL \\
STKREAL[cs,ms] &= \texttt{Hide}\ \{load, up, down, cdef, read, write, mdef, cdo\}\ \texttt{in} \\
&\quad CTR[cs] \parallel MEM[ms] \parallel SCTL
\end{aligned}
$$

Figure 12: Realization of a Stack

```
            (DI XOR$XOR$X CO1)
            (DI XOR$XOR$Y CO2)
            (DO (XOR-FN XOR$XOR$X XOR$XOR$Y) NXO)
            (DI MUX1$MUX$MX CDI1)))
NEXT-CTRL-STATE ((0 0 0 0 0 0))
NEXT-DP-STATE ((FF1$FF$X FF2$FF$X NIL NIL NIL NIL)))
```

A.7 One More Example: A Stack

From the definitions in figure 12, we deduced the definition in figure 13.

$$
\begin{aligned}
STKPAR[cs,ms] &= reset \rightarrow x = cdi \rightarrow STKPAR[x,ms] \\
&\mid push \rightarrow d = mdi \rightarrow \overline{idle} \\
&\quad \rightarrow STKPAR[up(cs), write(ms, read(cs), d] \\
&\mid pop \rightarrow \overline{idle} \rightarrow STKPAR[down(cs), ms] \\
&\mid top \rightarrow \overline{idle} \rightarrow STKPAR1[cs, ms, cs] \\
&\mid sdef \rightarrow STKPAR[cs, ms] \\
\\
STKPAR1[cs,ms,a] &= (mdo = read(ms,a)),\ STKPAR[cs,ms]
\end{aligned}
$$

Figure 13: $STKPAR$ Obtained via Parallel Composition

$$
\begin{aligned}
NCTL \;=\;& reset,\ \overline{mdef},\ \overline{cdef} \rightarrow \overline{load},\ \overline{mdef} \rightarrow NCTL \\
\mid\;& push,\ \overline{mdef},\ \overline{cdef} \rightarrow \overline{up},\ \overline{mdef} \rightarrow NCTL1 \\
\mid\;& pop,\ \overline{mdef},\ \overline{cdef} \rightarrow NCTL2 \\
\mid\;& top,\ \overline{mdef},\ \overline{cdef} \rightarrow \overline{cdef},\ \overline{read} \rightarrow NCTL \\
\mid\;& sdef,\ \overline{mdef},\ \overline{cdef} \rightarrow NCTL
\end{aligned}
$$

$$NCTL1 = \overline{write}, NCTL$$

$$NCTL2 = \overline{down}, NCTL$$

Figure 14: Realization of a Stack

A.8 A Pipelined Stack Controller

Figure 14 shows a pipelined stack controller. Using this controller makes the stack operate faster. (NCTL has more control states than SCTL though.) The notation "$\overline{write}$, $NCTL$" means: generate $\overline{write}$ during all the initial transitions of $NCTL$. We would like to automatically derive $NCTL$ from $SCTL$ by a technique that rearranges events without disrupting the meaning of the parallel composition. A suitable denotational model has been proposed by us for HOP to define process equivalence (see [GFK87]).