# 1 Introduction

The use of formal specifications for specifying, verifying, manually designing, and automatically synthesizing hardware systems is becoming widespread. Not only are there different formal specification languages, but also there are a number of different formalisms in use: Functional Programming [22, 38] Prolog [42], Petri Nets [10], Temporal Logic [5], various Calculii of Communicating Systems [30, 20] Trace Theory [39], Higher Order Logic [7, 23], Algebraic Specifications and Equational Techniques [18, 40, 35], Synchronized Transition Systems [13], and Path Expressions [2], to name a few. Enough impressive results have been demonstrated to justify the use of formal specifications for VLSI design. However, as will be discussed momentarily, many problems in the use of formal specifications for VLSI design remain unsolved. More importantly, many indirect benefits of studying and using formal specifications leading to the development of new kinds of VLSI design tools, for instance have not been emphasized enough.

In this paper, we present a simple and formal Hardware Description Language (HDL) "HOP" (Hardware viewed as Objects and Processes), and present its role in three VLSI design activities: (i) inferring concise behavioral descriptions of systems from their structural descriptions; this is done using an algorithm called PARCOMP; (ii) the detection of control timing errors during behavioral inference; (iii) productive and yet time efficient functional simulation using the inferred behavior. The main results presented are: (i) doing the above three tasks by capitalizing on the the formal semantic rules of the language HOP; (ii) demonstrating the utility of these ideas using a working implementation. HOP is continuously evolving, and this paper reports the status of our work as of (roughly) the end of 1988.

Despite being a formal specification language, HOP specifications can be easily understood by non theorists. HOP can capture with clarity intricate timing protocols that asynchronous hardware systems often exhibit. It has the ability to highlight timing/control aspects, and function/data aspects separately, so that designers may focus on one aspect at a time. Last, but not the least, HOP has a simple semantics that can be exploited for doing PARCOMP, functional simulation, and design verification. We now present the motivation for designing HOP, and some specific results.

## Motivation

Specifying the timing protocol and the functional behavior of asynchronous systems with clarity is quite important. Since functional details are often intricately interwoven with timing, separation of concerns is important. This has been achieved in HOP through the separation of function/data and timing/control aspects in the underlying semantics as well as in the text of the specification.

It has been reported that the complete formal verification of large IC designs is at present a demanding task [11]. Until this situation changes significantly, it appears that formal specifications will be used mainly for their indirect benefits: promoting better understanding of designs, better communication among hardware designers and systems software writers, and support for specification driven design automation activities. In this paper, we focus on such indirect benefits of using HOP.

**Specific Results Reported, and Organization of the Paper**

Section 2 illustrates HOP, and section 3 presents its operational semantics. Section 4 illustrates PARCOMP on a simple example, showing how each rule of the operational semantics is used. Section 5 presents various experiments conducted using PARCOMP. First, we present the result of performing PARCOMP on the stack module. We then deliberately introduce errors into the stack controller, and show how PARCOMP can reveal these errors. We then show how the stack may be pipelined, and present the behavior of the pipelined stack inferred using PARCOMP. We also show how PARCOMP can be used to make functional simulation more productive and efficient. Section 6 sketches a divide-and-conquer variant of the PARCOMP algorithm that works on regular arrays. Section 7 presents our conclusions; In appendix A.1, we briefly describe the HOP design system that was used to produce the results reported.

## 1.1 Understanding the Modeling Philosophy Behind HOP

One significant aspect of HOP is that it emphasizes the use of abstract data types for hardware modeling. Instead of specifying hardware systems as 'software abstract data types' and generating hardware designs from them (as done in *e.g.* [18, 13, 14]), we view hardware components as concurrent processes that internally use user-defined data types for conveniently modeling data path states. Instead of invoking data type operations to use data type objects (as done in software), HOP processes are used by providing events and data to them according to their *interface timing protocol*, specified via a finite control-state process. Operations on objects are not composed via functional expressions, as done while programming using data types; instead, the operators *parallel composition* and *hiding*, as introduced by Milner in CCS [32] and Hoare in CSP [21] are employed.

Consider a stack data type implementation that uses a counter to implement the stack pointer and a memory array to implement the stack locations. If such a stack were to be specified as a software data type, the definitions of the stack operations (say *push*, *pop*, and *top*) can be provided via functional expressions that use operators on the stack pointer and memory types. The stack state can be modeled as a tuple $< mem, ctr >$, consisting of the memory and counter states. The operation *push* can be defined as:

$$push(< mem, ctr >, v) \Leftarrow < write(mem, read(ctr), v), add1(ctr) > .$$

This says that the memory state should advance to $write(mem, read(ctr), v)$ and that the counter state should advance to $add1(ctr)$. This view of hardware systems—that they implement a collection of intuitive to grasp mathematical functions—is also taken in [22].

As we showed in our past work with the SBL language, such specifications may be implemented in hardware by synthesizing controller modules that 'fire' the operations *write*, *read*, *add1*, etc. in an applicative order (actually the *in situ evaluation order* [17], which is slightly more restrictive than the applicative order). However for this technique to be widely applicable, it should be possible to view a wide variety of hardware systems as data types. This isn't natural often, especially where control aspects dominate. More seriously, the 'software data type like' approach does not permit the specification of complex timings naturally, although it has been attempted [13, 37].

### The Concept of "Modes of Behavior"

HOP takes a crucial departure from the functional/data-type view of hardware. Rather than considering *data-type operations*, or *functions*, we focus on *modes of behavior*. A modes of behavior is like a *trace* [21]. A mode of behavior is best characterized as a finitely describable sequence of *events*, *data input actions*, and *data output actions*.

Consider a memory process modeled in HOP. Different realizations of the memory have different (depending on design decisions such as pipelining etc.) *read* modes of behaviors. One such mode of behavior consist of a *read* command event, a data input action corresponding to the supply of address, and a data output action corresponding to the output of the read data. These three actions may come in any order, with the only constraint that the $i$th read command event and $i$th address input must precede the $i$th data output. Clearly, many different modes of behavior are admitted by this (rather loose) constraint. For example, a memory with a pipelined implementation of the *read* operation defines one specific mode of behavior for read. A memory that queues up to (say) 12 read requests before it outputs any data item, defines another mode of behavior. So not only do we need mathematical functions to define I/O mappings from states and inputs to new states and outputs, but we also need a way to capture the timings involved. Functions and their mappings must be specified in conjunction with sequencing details. This extra degree of timing complexity is not well-handled by functional-programming based approaches for hardware modeling.

### Specifying Modes of Behavior in HOP

HOP is intended to capture modes of behavior directly. It does so by introducing a protocol specification section. Let us understand the way protocol sections are written. Consider the pipelined *read* operation, again. One of the most natural ways of explaining the behavior of such an operation is by drawing the picture of a Deterministic Finite-state Automaton (DFA) and associating a set of concurrent actions with each DFA transition. One may ask, "why not use classical DFAs for specifying hardware"?

This question is being considered mainly for two reasons. For one, in this paper we depict HOP process specifications using 'DFA-like graphs', and we want to avoid the readers mistaking HOP to be a DFA specification language. For another, it is known that explaining a new concept by first presenting a related but much weaker concept, and then showing that such a concept won't suffice, is very effective.

The following are some of the important reasons:

- DFA based languages cannot handle data related aspects well; modeling data path states as automaton states results in an explosion of the number of states. In contrast, in HOP we use high-level abstract data type (ADT) objects to model data related aspects. Only control states are explicitly modeled. Data aspects are captured by *annotating* the control graph. By doing so, both the data and control aspects of a system are completely specified at a high level.

- Hardware systems are developed over a long time, and initially, only the "what" aspects (*requirements*) on the system's behavior are known. High-level ADTs can be used to write a *requirements* specification of the system—and refined later when design details become known. These benefits are not available if DFA based models are used.

- HOP facilitates the writing of requirements specifications for the temporal aspects of a system using the concept of *events*.
- HOP's process model addresses design issues such as the connection of modules via busses, as well as the related issue of *strengths*[6]. It supports broadcast communication.
- HOP's process model is based on the three fundamental operations of hierarchical system design—*composition*, *hiding*, and *renaming*—as identified by Milner[32]. HOP is a high-level specification language for synchronous systems and provides a theoretical basis for the hierarchical design of VLSI systems.

## 1.2 Related Work

HOP is close in some respects to the work of Milne [29, 30]. The main differences are the following. In HOP, value communication has been decoupled from synchronization. The advantages of doing so are discussed in section 2. HOP processes are deterministic and lock-step synchronous, thus making it well suited for describing synchronous (synchronously clocked) hardware systems. A large majority of today's VLSI systems are synchronous. These decisions contribute directly to the simplicity of the language and makes specification driven design more practical. On the other hand, Circal primitives are more elementary. It is not specialized for lock-step synchronous systems.

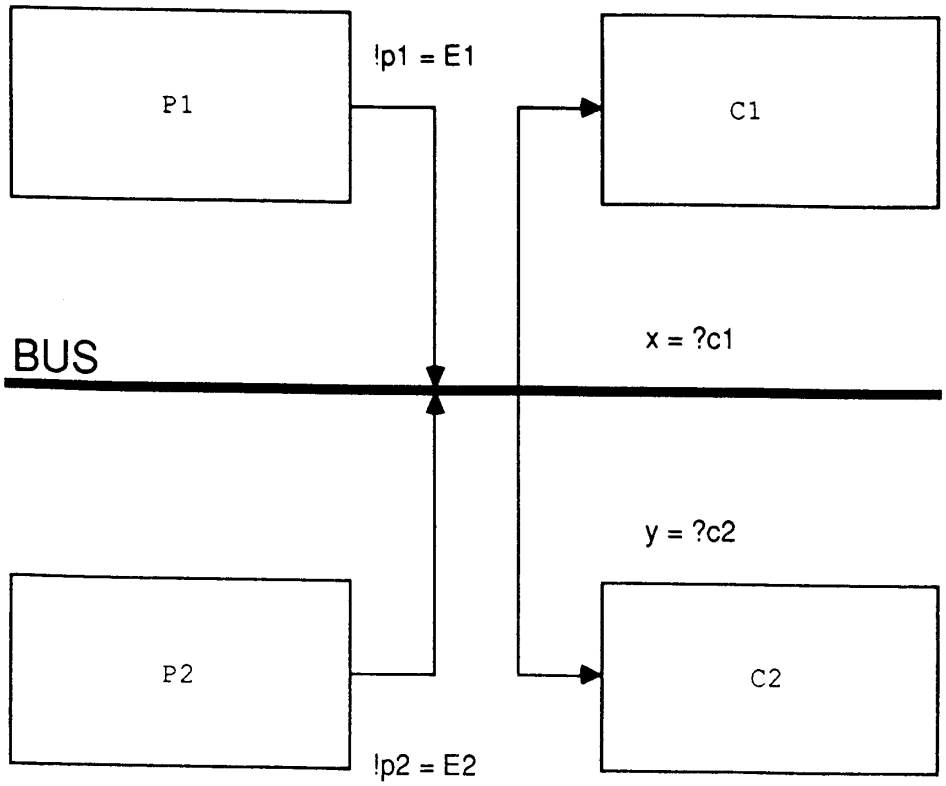
Compared to the LOTOS specification language [8], [26], HOP is much simpler, and differs in its adoption of a synchronous process model. The languages Esterel [4] and Lustre [9] are both based on synchronous process models, as HOP is. However, HOP is specialized towards hardware description and design, whereas both Esterel and Lustre are not. Neither PARCOMP, nor constructs similar to Vecprocs, exist in the above works, as far as we know.

HOP is different from more traditional languages (*e.g.* VHDL[41], Karl[33], and ISPS[3]) in many ways, the most important being the following: (i) it is much simpler, syntactically and semantically; (ii) it models hardware through a concurrent processes model—rather than through traditional programming constructs. This, in our opinion, captures hardware behavior more faithfully. Ideas developed in the context of domain specific and simple languages such as HOP can be easily migrated to VHDL.

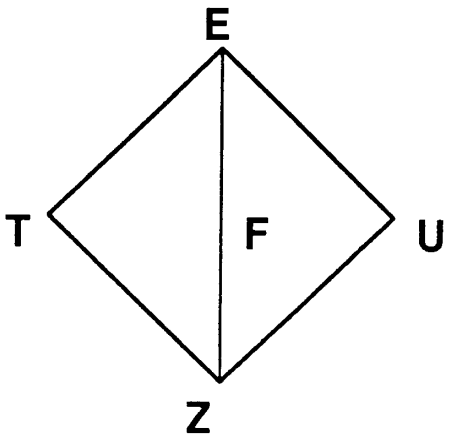
PARCOMP, as well as its planned uses, are similar to the work reported in [19], and to the idea of *constructive simulation* reported in [31]. However our work is done for a much higher level language that includes user-defined abstract data types. Our algorithm embodies useful static checks of timing protocols. Our algorithm capitalizes on the structural information (specifically, knowledge about events that are completely hidden within a module) to save on computation time. Further, we have developed a version of PARCOMP called PCDC (PARCOMP using *divide-and-conquer*) that can exploit the high regularity of many hardware systems to reduce its run-time.

Finally, PARCOMP can be used to save the time of simulation; we can perform a "pre simulation" of the tester and the testee using PARCOMP, and run the resultant process. These computational-effort saving measures are believed to be new.

# 2 The HOP Language




x of C1 = y of C2 = lub (E1, E2)




Lattice for computing lub

Figure 4

```
ABSPROC  <ModuleName> [<formal params pertaining to sizes & types>]
CONST    <list of constants of the same value>
TYPE     <list of type identifiers of the same type>
PORT     <list of ports of the same type; clocks are also ports>
EVENT    <events and their encodings in terms of port values>
PROTOCOL <a list of process definitions>
DEFUN    <a list of function definitions>
END      <ModuleName>
```

Figure 1: The Skeleton of an Absproc Specification

```
REALPROC   <ModuleName> [<formal params pertaining to sizes & types>]
CONST      <list of constants of the same value>
TYPE       <list of type identifiers of the same type>
PORT       <the external ports of the module being defined>
SUBPROCESS <instantiations of prev. defined abs/real/vec processes>
CONNECT    <the set of interconnections among the subprocesses>
END        <ModuleName>
```

Figure 2: The Skeleton of an Realproc Specification

```
VECPROC    <ModuleName> [<formal params pertaining to sizes & types>]
CONST      <list of constants of the same value>
TYPE       <list of type identifiers of the same type>
PORT       <the external ports of the module being defined>
SUBPROCESS <instantiations of prev. defined abs/real/vec processes>
DIMENSIONS <the SIZES of each dimensions of regularity>
CONNECT    <interconnections betn. subprocesses, via recurrence eqns.>
END        <ModuleName>
```

Figure 3: The Skeleton of a Vecproc Specification

<< Figure on separate sheet >>

Figure 4: Use of Data Assertions and Queries for Value Communication

The basic unit of specification in HOP is a *process*. A process is characterized either *behaviorally* or *structurally*. A process defined behaviorally is called an ABSPROC, standing for 'abstract process'. The internal structural details of an ABSPROC are not specified. Its behavior alone is specified by providing a protocol specification. Processes specified as a network of subprocesses are called REALPROCs, where subprocesses may themselves be ABSPROCs or REALPROCs. Since topologically regular realprocs (*e.g.* single and two-dimensional arrays of modules) occur very frequently in practice, we identify a sub-category of realprocs called VECPROCs. Vecprocs in HOP may best be regarded as "arhythmic arrays"—geometrically regular arrays in which computations aren't necessarily regular, or rhythmic, as in systolic arrays.

A process refers to many different pieces of information such as constants, ports, port types, functions, etc. It is convenient to package such definitions along with the protocol definition (for ABSPROCs) or the interconnection definitions (for REALPROCs) into a unit called a *module*. A module definition starts with the keyword ABSPROC, REALPROC, or VECPROC (as the case may be), followed by the name of the module, declarations, and ends with 'END modulename' (see figures 1, 2, and 3).

The PROTOCOL section of an ABSPROC lists a collection of PROCESSes that are defined mutually recursively. One process in this collection is identified to be the 'top level process'—one that specifies the behavior of the hardware unit at the time when it is 'powered up'. The name of the top level process is the same as the name of the module.

## 2.1 Specifying an Absproc

An absproc is specified by its ports, its events, and its protocol. We now examine some of the unconventional sections of an absproc specification in each of the following subsections.

### 2.1.1 Ports and Value Communication

The mechanism of synchronized communication, such as used in *e.g.* [21] or [30], does not accurately model the value communication in hardware systems. Two aspects that are not satisfactorily modeled by these models are: combining values of different strengths; and broadcast communication.

As an example, consider figure 4 which depicts a system consisting of two producer processes $P1$ and $P2$ that can communicate with two consumer processes $C1$ and $C2$ over a bus. In general there can be many producers as well as consumers. In such systems, it is perfectly acceptable to have all the following situations: one producer alone writing while a consumer is reading; more than one producer writing; a consumer seeking the value on the bus while there are no simultaneous producers; etc. The values deposited on the bus by the various producers are first 'combined' and then 'broadcast' (instantaneously made available to all consumers) before the next clock arrives.

In HOP, Value communication is performed through mechanisms called *data assertions and queries*. A data assertion, written as `!p=E`, asserts that output port `p` is carrying the value denoted by the functional expression `E` at the time of the assertion. The data assertion `!p=Z` (Z stands for high-impedance) can model the output of tristated drivers. For modeling a pull-up transistor, the assertion `!p=weak1` may be used.

A data query, written as `x=?q`, binds `x` to the value of *input port* `q` at the time the query is made. For handling multiple data assertions, the type of values communicable via ports must be organized into a strength lattice, *e.g.* [6]. Multiple data assertions (as in bus connections) then end up asserting the *least upper bound* with respect to the strength lattice of the asserted values, on the port. For example the *bit type* of HOP includes the weakest value Z (*high-impedance*), truth values T and F, an unknown value U, and the most dominant value E, *error*. T,F, and U are incomparable amongst themselves and lie in-between Z and E. This lattice may also contain other values, such as `weak1` and `weak0`.

We now provide two examples to clarify all this. A wired-or connection can be modeled as one producer of a value `weak1` (the pull-up resistor) and some producers of `strong0` (signifying the pull-down paths). As another example, P1 outputs a value 'Z' (high impedance) while P2 is a flip-flop that has just been powered up, and hence outputs the 'U' value (unknown); the result will be the *least upper bound* of Z and U, namely U. This value will be obtained by C1 and C2, if they were to query the bus at this time.

In HOP, we can model only unidirectional interactions. I/O ports may be bidirectional in the sense that only one direction prevails at a time.

### 2.1.2 Events

Events are of two kinds: input, and output. An input event `e` (written `Ie`) denotes a (boolean valued) *condition* that a process awaits (or *relies upon*) as a precondition to doing some computation. An output event `e` (written `Oe`) denotes a condition that a process generates. Usually a process has many possible computations to pursue and it selects one of these based on input events. An input event has to *synchronize* in the sense that it must be matched by a corresponding output event generated by another process, if the computation conditioned upon the input event is to actually take place. Output events do not have to synchronize, in the sense that it is permissible to have one process generating an output event (say, by asserting a control line) without having any other processes simultaneously awaiting this event as an input event.

This lack of symmetry between input and output events is another aspect that differentiates our work from works such as CCS, CSP, and Circal. Our reasoning behind this choice of semantics is the following. In designing a module M, it is not possible to anticipate the behaviors of the communicating partners of M. Therefore, providing more information from M (by generating more output events than necessary) should not make a difference as far as the communicating partners of M are concerned because they can ignore these additional output events.

Events provide yet another form of abstraction. While designing a hardware module, a designer quite often anticipates the need to generate/sense certain conditions, even before knowing *how* such conditions are to be encoded. Viewed this way, we can talk about *event connections* between hardware modules rather than (the more detailed) *control wirings*. The

actual implementation of event connections requires combinational logic (often called 'glue logic') that translates a condition in one module to a condition in another.

Finally there is yet another important use of events—*abstract events* that are merely meant to highlight interesting points in time. Abstract events are not realized in hardware. The significance of abstract events is illustrated by the following generic example. In traditional designs of synchronous systems, the completion of an operation is often not explicitly notified, but is tacitly assumed after the elapse of a certain interval of time from the start of the operation. Such hard-wired delays are compared to hard-wired literal constants in programs that are known to lead to programs that are hard to debug or modify. By highlighting points in time through events, we achieve two goals: (a) specifications become more readable; (b) PARCOMP is better able to match events across communicating modules and thereby discover sequencing errors; since PARCOMP is run prior to simulation, we effectively have a 'temporal type checking system' that can detect sequencing errors, much like type-checking avoids certain run-time errors.

#### Synchronization and Communication

By having two processes interaction mechanisms (*events* and *data assertions*) we have essentially separated synchronization from communication. The key idea is that due to lock-step synchronous execution, processes can implicitly synchronize by monitoring their own execution rates, and thereby exchange data across (memoryless) wires by being "in the right state at the right time". For example, consider a counter with two commands *reset* and *up* that are triggered via events with the same names. The counter can, after it has been subject to the *reset* event and until it is subject to the *up* event, assert 0 on its output port. Process that are responsible for the *reset* and the *up* events can rely on this fact, and query the counter's output in between the execution of the *reset* and *up* events. Such queries go 'completely un-noticed' by the counter itself—i.e. it doesn't have to rendezvous. This style of specification achieves the effect of value broadcast quite naturally, as data assertions reach wherever the port that carries the data is connected. Also, the above features lead to writing modular specifications for two reasons: (i) the specification writer for counter does not have to know how many times the counter will be queried in between a *reset* command and an *up* command; (ii) we do not have to simulate broadcast by using multiple unicast; doing so would require knowledge of the number of recipients, a priori.

### 2.1.3 Boolean Guards

Events are conditions generated *external* to a module, and are like propositional variables. Sometimes, while modeling a system it is not possible to specify all conditions of interest using events. We therefore introduce another construct called *boolean guards*. Boolean guards are expressions involving predicates over data path states and data inputs. Example: in a bounded stack, a boolean guard $not(full(stack))$ will be a pre-condition for the application of a *push* operation.

It is computationally far easier to check for events awaited in a process against events asserted in another process to see whether the awaited events are satisfied, because all this involves is the syntactic comparison of the names of events. In comparison, it is much more difficult to statically simulate the effect of value communications between modules and to then

determine which boolean guards are true when. Therefore, by expressing as many conditions as events, the HOP system is able to more easily prune away unrealizable modes of behaviors during PARCOMP. In addition, many conditions such as command inputs and status signals are easy to model using events.

For these reasons, we encourage HOP programmers to identify as many conditions as events, and use boolean guards only for more complex situations, or when the number of conditions to be modeled is too large.

### 2.1.4 Data Path States

In the specification of an absproc, the data path state of the system being specified can be modeled using an appropriate high-level ADT. In our experience (*e.g.* [15]), the use of ADTs having simple definitions can make reference specifications far more reliable and easier to understand.

### 2.1.5 The Timing Model

HOP is based on the model of *conservative clocking*, which means that each clock period accommodates the delays of all the combinational stages whose evaluation the clock initiates. In such systems there are two measures for time: clock period, and the delays of combinational stages. Many low-level simulators treat the clock as 'yet another signal', and hence cannot capitalize on many nice properties exhibited by conservatively clocked systems. For instance, in a conservatively two-phase clocked system, phase-1 and phase-2 initiated events alternate in time.

Another important optimization is possible if combinational loops are absent *and* the system is conservatively clocked. In such systems it is possible to set all combinational delays to zero and obtain an accurate *functional simulation model*; this model accurately predicts the behavior of systems over clock periods. To better understand the role of the above assumptions let us relax them one at a time. If conservative clocking is used but combinational loops are allowed, a fixed-point [27] computation, as done by (say) MOSSIM [6] will be needed to find out the system state after one clock period. If conservative clocking is not used, the behavior can be predicted only by a more complex procedure, such as a timing simulator that maintains time-sorted event lists.

In HOP, we assume both conservative clocking and the absence of combinational loops. The latter fact can be checked by HOP's simulation preprocessor by analyzing, for each (clock) time step, whether a data assertion expression depends upon itself. With these assumptions, HOP implements the Huffman model of hardware where the current inputs and state *functionally* determine the current outputs and the next state. Inputs and outputs consist of events and data, and the state consists of the control and data-path states. Roughly speaking, HOP relates to the Huffman model in a way similar to how Pascal relates to Turing machines.

### 2.1.6 An Example of an Absproc: A Pipelined Memory

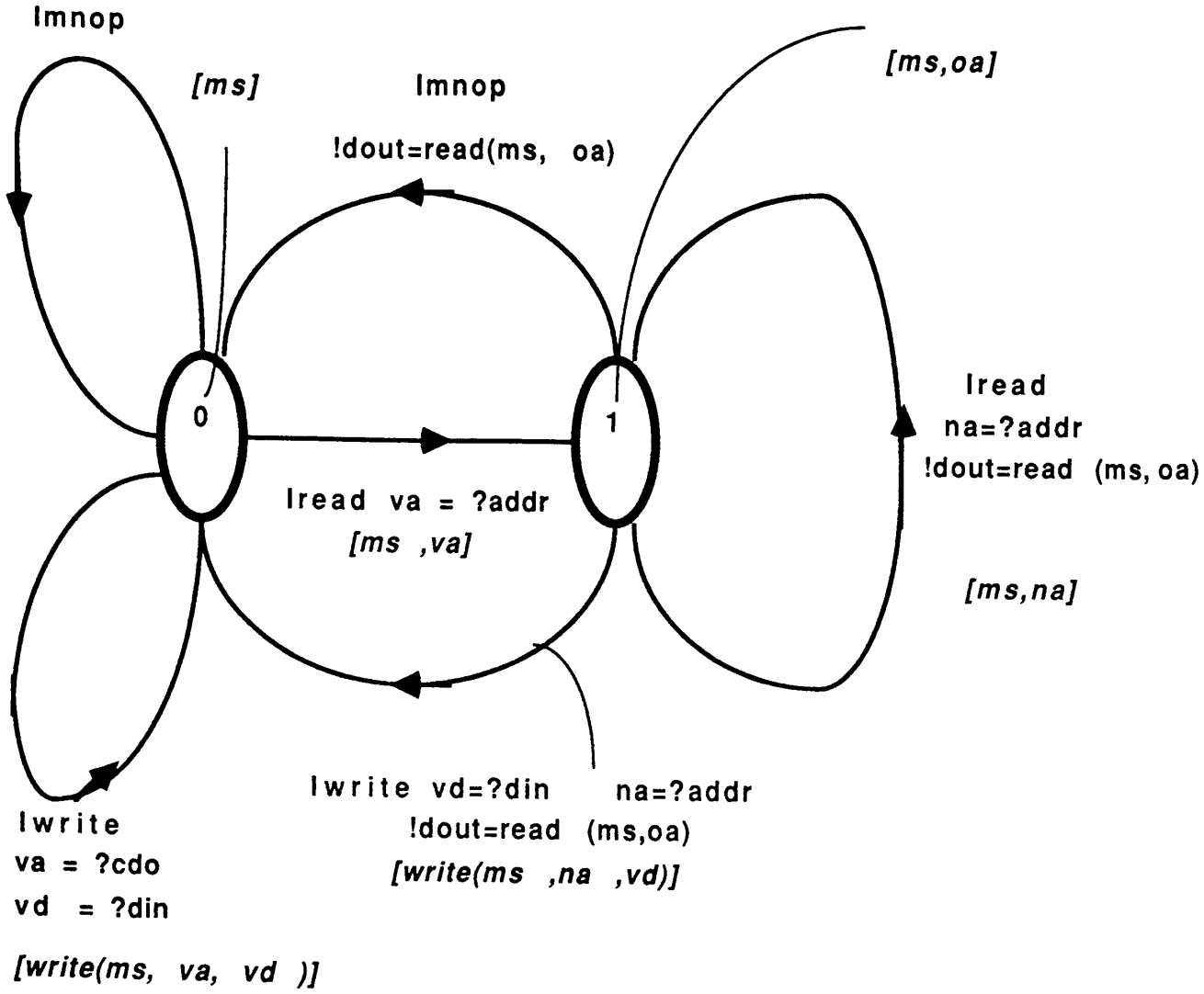
Consider memory module `MEM` which has an address input port `?addr`, a data input `?din` port, and a data output port `!dout`. It can, in its 'powered up' state, entertain events `Imnop`, `Iwrite`, and `Iread`, which implement (respectively) the commands mnop (memory's no op),

```
-- This is a comment.
ABSPROC MEM [ address_size, data_size : int ] -- Note-0
TYPE
 addressType = 0 .. address_size - 1
 dataType    = 0 .. data_size - 1
 memoryType  = array[addressType] of dataType
PORT
 ?din, !dout : array [data_size] of bit
 ?ain : array [address_size] of bit
EVENT
 Imnop, Iread, Iwrite = TBD
PROTOCOL
      MEM  [ms : memoryType] <=
                Imnop -> MEM [ms]
              | Iwrite, va=?addr, vd=?din -> MEM [write(ms,va,vd)]
              | Iread, va=?addr -> MEM1[ms, va]     --^-- Note-1

      MEM1 [ms : memoryType, oa : addressType] <=
                Imnop, !dout=read(ms,oa) -> MEM [ms]
              | Iwrite, na=?addr, vd=?din,
                        !dout=read(ms,oa) -> MEM [write(ms,na,vd)]
              | Iread, na=?addr, !dout=read(ms,oa) -> MEM1[ms, na]
DEFUN
 write :: m : memoryType, a: addressType, d:dataType -> m1 : memoryType
                IF (> addr memSize)
                   (print "Illegal memory address")
                   (error-obj memType)
                ELSE (update-vector memType m a d)       -- Note-2

 read :: m : memoryType, a: addressType -> d : dataType
                IF (> addr memSize)
                   (print "Illegal memory address")
                   (error-obj int)
                ELSE (index-vector memType m a)          -- Note-2
END MEM
-- Note-0 : Upper and Lower Cases are Treated the Same in HOP.
-- Note-1 : write (defined in DEFUN) computes the new data path state.
-- Note-2 : index-vector and update-vector supported by memoryType
```

Figure 5: Specifications of a Memory

<< Figure on separate sheet >>

Figure 6: Depiction of the PROTOCOL Specification of MEM




STATE 0 ---> MEM

STATE 1 ---> MEM1

Figure 6

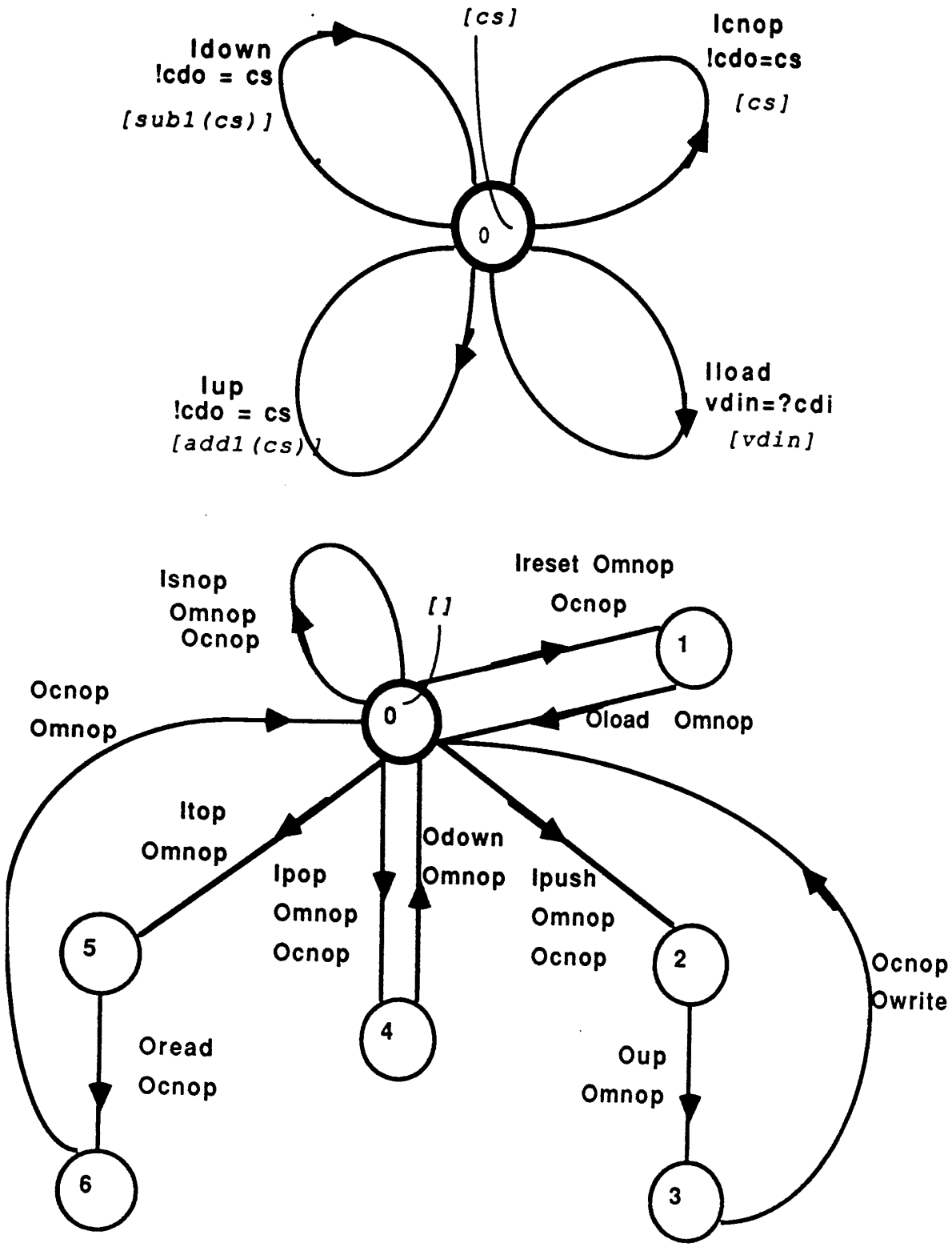

[cs]
Idown
!cdo = cs
[sub1(cs)]
Icnop
!cdo=cs
[cs]
0
Iup
!cdo = cs
[add1(cs)]
Iload
vdin=?cdi
[vdin]
Isnop
Omnop
Ocnop
[]
Ireset Omnop
Ocnop
1
Ocnop
Omnop
0
Oload Omnop
Itop
Omnop
Ipop
Omnop
Ocnop
Odown
Omnop
Ipush
Omnop
Ocnop
5
2
Ocnop
Owrite
4
Oread
Ocnop
Oup
Omnop
6
3

Figure 7

```
CTR [cs] <=   Icnop, !cdo=cs    -> CTR [cs]
            | Iload, vdin=?cdi  -> CTR [vdin]
            | Iup, !cdo=cs      -> CTR [add1(cs)]
            | Idown, !cdo=cs    -> CTR [sub1(cs)]

SCTL <=   Isnop,  Omnop, Ocnop  -> SCTL
        | Ireset, Omnop, Ocnop  -> Oload, Omnop -> SCTL
        | Ipush,  Omnop, Ocnop  -> Oup, Omnop   -> Owrite, Ocnop -> SCTL
        | Ipop,   Omnop, Ocnop  -> Odown, Omnop -> SCTL
        | Itop,   Omnop, Ocnop  -> Oread, Ocnop -> Omnop, Ocnop  -> SCTL

-- All the ''nop'' events have to be specified in the present version of HOP.

                << Also see figure on separate sheet >>
```

Figure 7: Stack's Submodules:- CTR: An up/down counter; SCTL: Stack Controller

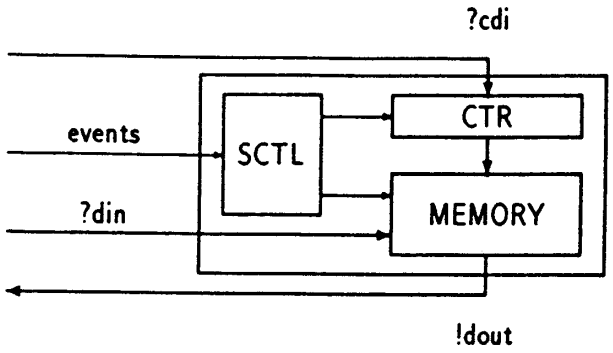



Figure 8: Schematic of the Realproc of a Stack

```
REALPROC stack [<various size & type parameters>]
PORT
  ?cdi, ?din, !dout : <suitable types>
EVENT
  Ireset, Ipush, Ipop, Itop, Isnop = TBD
SUBPROCESS                                          -- Note-3
  MEM : mem [<actual size parameters>]
  CTR : ctr [<actual size parameters>]
  SCTL : sctl
CONNECT
  DATANODE
    HIDDEN CONNECTS ((MEM ?ain) (CTR !cdo))      -- Note-1
    ?din   CONNECTS ((MEM ?din))                   -- Note-4
    ?cdi   CONNECTS ((CTR ?cdi))
    !dout  CONNECTS ((MEM !dout))
  EVENTNODE
    HIDDEN CONNECTS ((MEM Imnop)  (SCTL Omnop))   -- Note-1
    HIDDEN CONNECTS ((MEM Iread)  (SCTL Oread))
    HIDDEN CONNECTS ((MEM Iwrite) (SCTL Owrite))
    HIDDEN CONNECTS ((CTR Icnop)  (SCTL Ocnop))   -- Note-2
    HIDDEN CONNECTS ((CTR Iload)  (SCTL Oload))
    HIDDEN CONNECTS ((CTR Iup)    (SCTL Oup))
    HIDDEN CONNECTS ((CTR Idown)  (SCTL Odown))

    Ipush  CONNECTS ((SCTL  Ipush))               -- Note-4
    Ireset CONNECTS ((SCTL  Ireset))
    Ipop   CONNECTS ((SCTL  Ipop))
    Itop   CONNECTS ((SCTL  Itop))
    Isnop  CONNECTS ((SCTL  Isnop))
END stack

--Note-1: These are hidden ports/events
--Note-2: Currently we have to specify even ''obvious defaults''; eg. Ocnop.
--Note-3: Module instance names and module type names are different, in general.
--Note-4: The following ports/events are not hidden.
```

Figure 9: Realproc of a Stack

write, and read. MEM is pipelined thus: the delivery of the result of a read request is overlapped with waiting for the next command. Operation write as well as operation mnop (no operation) aren't pipelined.

Let us study figure 5. The header declares two size parameters. The PORT section declares the I/O ports. The EVENT section defines three events, and equates them to "To Be Defined" (TBD). Thus, the designer of MEM doesn't yet care about the encodings of the control inputs as well as clocks (if any). He/she pretends that Iwrite, Iread, and Imnop are three bit wires coming in.

Consider the PROTOCOL section. This section can be depicted as shown in figure 6. This is because HOP processes are finitely representable processes (that is, they have a finite-state control skeleton, and this control skeleton can be annotated ("decorated") with data path state changes and port value assertions.) These annotations are done in a purely functional notation.

The functional expressions used in the PROTOCOL section are defined in the DEFUN section and/or in the ADT library. The ADT library is implemented using object oriented techniques (our technique: "generic types are classes"), using the language FROBS [24]. Therefore, function definitions are realized as *overloaded* methods, that are dispatched correctly. Besides, subtyping is easy to support via class inheritance. Many of the data types of HOP support both immutable and mutable constructors. This is to support the implementation of the *in situ evaluation* technique [17] to use mutable constructors whenever possible, while preserving the referential transparency of HOP functional expressions.

Let us study the text of the PROTOCOL section. This section is also depicted in figure 6. In this figure, we have *annotated* the transitions with current events, data queries and assertions, and the next data path state. (The next data path state is shown only if it is different from the current data path state.)

Process MEM begins in control state `MEM` and in datapath state `ms`. It offers a choice of three events, `Imnop`, `Iwrite`, and `Iwrite`. If none of these events is asserted externally, the behavior of `MEM` is undefined. Event `Imnop` causes `MEM` to go back to control state `MEM`. Event `Iwrite` when asserted from outside must be accompanied by data assertions `va` on the `?addr` bus, and `vd` on the data bus `?din`. `MEM` goes back to the control state `MEM`; however its datapath state changes to `write(ms,va,vd)`. Event `Iread` must be accompanied by a data assertion `va` on port `?addr`. The next control state attained is `MEM1`, and the next data path state is a pair `[ms,va]`.

In control state `MEM1`, process `MEM1` is in data path state `[ms,oa]`. It again offers the choice of three events. However note that while waiting here, the data assertion `!dout=read(ms,oa)` is made. This assertion corresponds to the result of the *previously* requested `read`. (This is the pipelining effect). While `reads` keep coming, `MEM1` goes back to `MEM1`. A `Iwrite` or `Imnop` takes `MEM1` back to `MEM`.

If this memory were to be used in a clocked system, the events `Iwrite`, `Iread`, etc. would be generated at the appropriate clock phases. Details such as multiphase clocking would be described in the EVENT section of an ABSPROC by replacing the "TBD"s by boolean expressions involving input control wires and clocks.

In HOP, the capability of a module to 'idle' must be explicitly modeled. Events such as `Imnop` serve this purpose. With this view, every HOP module has to be executing at least one of its operations at every time step.

## 2.2 Specifying Realprocs and Vecprocs

A realproc is built using one or more absprocs by connecting some ports and events of the absprocs, composing the external protocols of the absprocs using the || operator, and internalizing (hiding) some of the events and ports. A syntactically sugared notation (`DATANODE` and `EVENTNODE` ) mitigates the burden of specifying the *renaming* and *hiding* [32] information. A Vecproc is essentially built in the same fashion; however a notation based on recurrence relations is provided to easily specify the regular placement of modules as well as regular interconnections among them.

A realproc specifies a system's realization. As an example let us use the memory unit in figure 5 to build a stack using an absproc `CTR` to implement the stack pointer and a controller `SCTL` to control the stack. The design of the stack would be specified by writing a realproc specification, as shown in figure 9. This specification captures the schematic shown in figure 8.

In the PORT and EVENT sections, the *external* ports and events of the realproc are declared. All other ports and events are assumed to be *internal*, and hence hidden from the outside world.

In the SUBPROCESS section, previously specified abs/real/vec processes are instantiated to the required sizes as well as types. For example we could now instantiate a generic stack to be a stack over bytes. The subprocesses themselves are described in figure 7. (We present only the PROTOCOL section of the subprocesses.)

In the CONNECT section, interconnections between: (i) ports and events of the submodules, and (ii) between the submodules and the external ports/events of the parent module, are specified. Semantically, connections are treated as *renamings*, in the style of [32]. For example, if we have a set of connected ports $P$, every $p_i \in P$ is renamed to $p_{new}$, a new name.

Let us look at the first two lines of the DATANODE subsection of the CONNECT section. The node that connects `?ain` of MEM and `!cdo` of CTR is hidden. The `?din` port of MEM connects to `?din` of the stack.

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

Figure 10: Abstract Syntax of HOP

# 3 Semantics of HOP

## 3.1 An Operational Semantics for HOP

We first briefly explain the translation from the user-level syntax of HOP to its abstract syntax given in figure 10. We then present the operational semantics.

### 3.1.1 Translation Into the Abstract Syntax

The PROTOCOL section of an ABSPROC definition consists of processes that are defined mutually recursively. Each process (such as SCTL in figure 7) defines the behavior of the hardware module over intervals of time whose lengths can be greater than 1. These processes are first translated into an equivalent, but much larger collection, of simpler processes, each of which describes the behavior over one unit of time. Each process in the translation is of the form:

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

where the 'arms' of the choice are subscripted using $i$. Such a translation simplifies the specification of the semantics, as well as the implementation of HOP.

In each such process $ie_i$, $dq_i$, $g_i$, $oe_i$, and $da_i$ are (respectively) sets of input events, data queries, boolean guards, output events, and data assertions for the $i$th arm of the choice. Determinacy requires that only one arm of the choice must actually be chosen during execution: that arm where all $ie_i$ synchronize and all $g_i$ are true. This is decided by the communicating partners of process $P$, as will be explained shortly.

A REALPROC is translated into the abstract syntax by first creating copies of the SUBPROCESS instances and renaming the local names within the ABSPROCs of the subprocesses. Now the CONNECT section is processed as explained below:

- The construct `HIDDEN CONNECTS ((SM1 port1) (SM2 port2) ...)` captures the fact that `port1, port2, ...` are all connected to a common point which is then hidden. `Port1, port2, ...` are renamed to `newportname`. `Newportname` is then recorded as being *hidden*—a fact that will be used below.
- The construct `externalport CONNECTS ((SM1 port1) (SM2 port2) ...)` captures the fact that `port1, port2, ...` are all connected to a common point which is then exported via `externalport`. All these ports are renamed to `externalport`.
- Connections among events are translated in the same way as the connections among ports, as done above.
- Unconnected ports and events are renamed to completely distinct names.

- Output ports meeting at a node $N$ are turned into connections into the input of a BUS module that applies the *lub* function to its inputs; the output of the BUS module is connected to node $N$. Because of the introduction of these fictitious BUS modules, we do not have to consider the case of more than one output port meeting at a node, in the semantics.
- Subprocess instances within a REALPROC start at their 'top level' control states and march in unison, interacting via their common events and ports, some of which are hidden outside the REALPROC. This interaction is captured in HOP's semantics by the || operator and the 'Hide x in P' construct. Thus, a REALPROC is translated into a construct of the form

  Hide $ie_i$ in ... Hide $oe_j$ in ... Hide $?p_k$ in ... Hide $!p_l$ in ...

  Subprocess1[vars1] || Subprocess2[vars2] ||... . It is assumed that the names used within each of the subprocesses of the REALPROC are renamed so as to be distinct, to handle scoping rules.

## 3.2 Operational Semantic Rules

The operational semantic rules are defined via structural induction over the abstract syntax, as done by Plotkin in [36]. We will consider a process $P$ at time $t$, and define the relation

$$P[E1](t) \xrightarrow{ie(t),dq(t),g(t):oe(t),da(t)} P'[E2](t+1)$$

where $ie(t), dq(t), g(t) : oe(t), da(t)$ are the possible sets of actions of $P[E1]$ at time $t$. This means that an external agency has to supply $ie(t), dq(t), g(t)$ in such a way that $ie(t)$ is synchronized by a matching $oe'(t)$, $dq(t)$ by a $da''(t)$, and $g(t)$ becomes true. Then only will the possible behavior really manifest. From here onwards we omit $t$ and $(t+1)$.

### 3.2.1 Rule for Deterministic Choices

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

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

### 3.2.2 Rule for Parallel Composition

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

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

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

- $IE(ie, oe) = ie \setminus oe$; those $ie$ that are not 'satisfied' are left over. Intuitively, we are symbolically simulating the subprocesses of a REALPROC, and are determining statically those events that are awaited by some hardware units which are at the same time furnished by other hardware units. Input events that are awaited but not asserted are retained, because when a 'third module' is brought into consideration, it may well be satisfied. Input events that are matched by corresponding output events are removed because they are satisfied by the output events, and hence need not be awaited.

- $DQ(dq, da)$ returns every $(q = ?p) \in dq$ for which there is no corresponding $(!p = E) \in da$. Again we are symbolically studying the interaction among the various data queries and assertions. 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 path state of the processes change as a result of value communications between processes.

### 3.2.3 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.

### 3.2.4 Determinacy of Choices

The syntax and the semantics introduced thus far does not prevent the definition of processes for which the transition relation is not a function. For instance, it is possible to define a process $P$ such that $P[dps] \xrightarrow{ie} P1[dps]$ and $P[dps] \xrightarrow{ie} P2[dps]$. On the other hand, HOP is supposed to capture the behavior of synchronous hardware systems that are deterministic.

We can render HOP deterministic by imposing syntactic restrictions on its *choice* construct (section 3.2.1). Determinacy will be achieved if the truth value assignments for the events and guards are such that only one arm of the choice is selected. This can be achieved by obeying the following restrictions on events and guards:

- If the set of input events on the $i$th arm of the choice is contained in the set of input events on the $j$th arm of the choice ($i \neq j$), then $g_i \wedge g_j$ must be false. In other words, if we consider input events alone and find that two arms of the choice qualify, then the boolean guards must allow only one of the transitions to be actually taken.

- For those pairs of transitions where the set of input events on the $i$th arm of the choice is not contained in the set of input events on the $j$th arm of the choice, and vice versa, ($i \neq j$), the guards need not be mutually exclusive. In this case, define an event $choose_i$ to be the conjunction of the events in $ie_i \setminus ie_j$; similarly define $choose_j$ to be $ie_j \setminus ie_i$; then, $choose_i$ and $choose_j$ must be mutually exclusive. Example: If there are three transitions for $P[dps]$ via input events $(ie1, ie2)$, $(ie2, ie3)$, and $(ie1, ie3)$ respectively, then $ie1$, $ie2$, and $ie3$ must be mutually exclusive of each other. This information can be passed on to the design phase that is responsible for implementing events (say, using boolean encodings of control wires and clocks).

If boolean guards are expressions over a decidable subset of first order logic (which they are, usually), determinacy can be statically checked.

# 4 Illustration of PARCOMP

## 4.1 What Exactly Does PARCOMP Do?

PARCOMP takes as input a realproc or a vecproc and produces as output an absproc. The absproc inferred by PARCOMP captures, via symbolic expressions, the behavior of the realproc

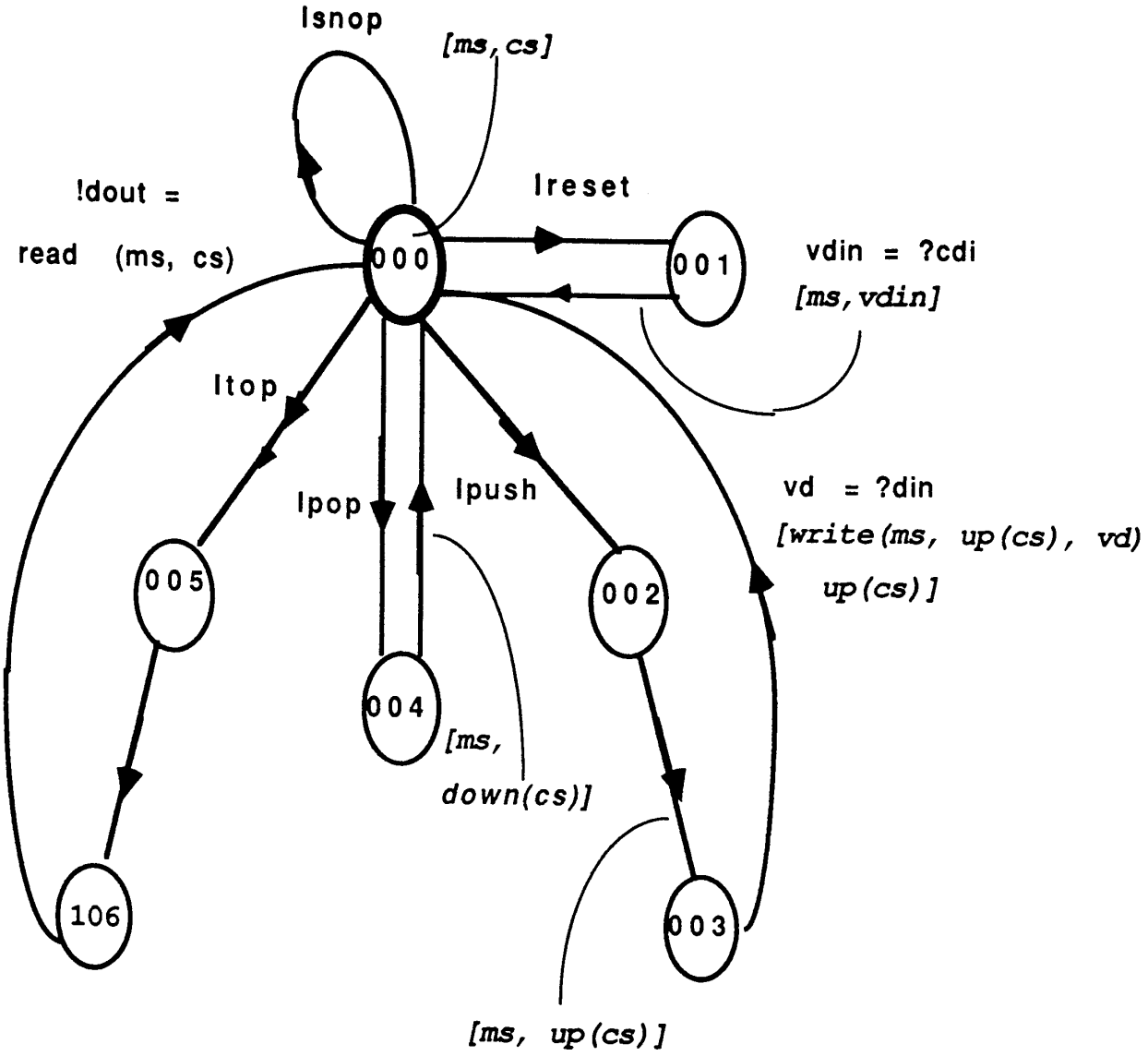

!snop
[ms, cs]
!dout =
read (ms, cs)
!reset
000
001
vdin = ?cdi
[ms, vdin]
!top
!pop
!push
vd = ?din
[write(ms, up(cs), vd)
up(cs)]
005
002
004
[ms,
down(cs)]
106
003
[ms, up(cs)]

Figure 11

```
PROTOCOL
STACK [cs,ms] <=
     Ireset -> di = ?cdi ->  STACK [di,ms]
   | Ipush  -> Oidle -> vd=?din ->  STACK [add1(cs), write(ms,add1(cs),vd)]
   | Itop   -> Oidle -> !dout=read(ms,cs) ->  STACK [cs,ms]
   | Ipop   -> Oidle ->  STACK [sub1(cs), ms]
   | Isnop  -> STACK [cs,ms]
                  << Also see figure on separate sheet >>
```

Figure 11: Absproc Automatically Inferred from stkreal using PARCOMP

or vecproc for all possible starting data-path states of the submodules, and for all external inputs. The text of the inferred absproc can be manually studied to see if the system behaves as understood by the designer. Thus, PARCOMP can greatly facilitate the understanding of the collective behavior of a collection of synchronous systems.

In the inferred behavioral description, PARCOMP does not retain any of the unused capabilities of the system. Consider a system built using three modules A, B, and C, where C is the controller for A and B. Though A and B may individually support (say) 5 operations each, C may actually use only (say) 2 each of their operations. In addition, C may sequence these two operations only in a small number of ways—out of the large number of possible ways in which they may be sequenced. PARCOMP discovers and retains only these modes of behaviors by capitalizing on the *event hiding* information supplied by the designer.

The use of the event hiding information not only makes the inferred behavioral description concise, but also reduces the run-time of PARCOMP. The worst-case time complexity of PARCOMP is proportional to the number of control state tuples (tuples of the control states of the subprocesses) actually generated. By pruning away control state tuples as early as possible, entire control subgraphs are eliminated quite early during the execution of PARCOMP. As a concrete example, in a pipelined memory system, PARCOMP generated 720 initial moves to explore but immediately discarded 719 moves as they had hidden unsynchronized events. This also implied that PARCOMP was not recursively invoked on those states that were reachable only through those 719 moves.

## 4.2 Illustration of PARCOMP on the Stack

Given the above stack realproc specification and given the specifications for `CTR` and `SCTL` shown in figure 7, we can use PARCOMP to infer the equivalent absproc specification `STACK` shown in figure 11. (Only the PROTOCOL section of the inferred process is shown. Inferring the behavior of the stack takes less than ten seconds of elapsed time running on a 1-MIP workstation, running Common Lisp.)

The inferred PROTOCOL specification asserts that the STACK system offers a choice of events `Ireset`, `Ipush`, `Itop`, `Ipop`, and `Isnop`. Let us study `Itop`. After asserting this event, the external world (say, the "tester process" of the stack) has to idle for one tick. No event is entertained by the stack (signified by the absence of any input events following `Itop`), as it is internally busy. (The system puts out the `Oidle` event when no user-declared event or data I/O is occurring.) During the second tick, it asserts the data value $read(ms, cs)$ on the `!dout`

port. This symbolic expression confirms that the stack would output the correct result on port `!dout` following the *top* command. Finally, the `STACK[cs,ms]` process continues to behave like `STACK[cs,ms]` itself, meaning that the STACK process did not undergo any datapath state changes.

Let us study the *push* operation. The external world is expected to supply the item to be pushed *two ticks* after it applied the `Opush` trigger that matched with the `Ipush` event. If this value were `vd`, then the future behavior of STACK would be like that of STACK[add1(cs),write(ms,add1(cs),vd)]. This symbolic expression shows that the *push* operation was implemented correctly. This is because the counter state has advanced from `cs` to `add1(cs)`, and the memory state has advanced from `ms` to `write(ms,add1(cs),vd)`. Informally, the stack pointer was incremented, and the memory location pointed to by the new stack pointer was written with `vd`.

The other operations can be similarly studied. It may be noticed that modes of behavior such as 'write into the memory and then increment the stack pointer' are not generated, though the counter as well as the memory individually are capable of this mode of behavior. This is because controllers decide what happens in a system, and the stack controller doesn't orchestrate the submodules in many (illegal) ways.

## 4.3 How Does PARCOMP Work?

### 4.3.1 Lockstep Cartesian Product

Our explanation of PARCOMP would be greatly facilitated by introducing the concept of *lockstep cartesian product* (LCP). Given two DFAs A and B, the 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 in $lcp(A, B)$. Further, the edge $EF_{ij}$ is in $lcp(A, B)$ directed from $< A_i, B_i >$ to $< A_j, B_j >$. $EF_{ij}$ is labeled with the union of the annotations on $E_{ij}$ and $F_{ij}$.

Example: View the process diagrams in figure 12 as DFAs, with state 0 as the starting state. Then, $lcp(A, B)$ contains all the 25 states in the cross-product of A and B. On the other hand if the loop from state 0 to state 0 of process B is absent, $lcp(A, B)$ will contain only the states 00, 11, 22, 33, and 44. The edges in $lcp(A, B)$ would then be: $00 \rightarrow 11$, $11 \rightarrow 22$, $22 \rightarrow 33$, $33 \rightarrow 44$, $44 \rightarrow 00$. Thus, in general, the number of states in the *lcp* is less than or equal to the number of states in the cartesian product of the states of the constituent processes.

PARCOMP is an algorithm that embarks on creating the LCP, and in the process begins to clash the annotations on the transitions of the subprocesses involved. In doing so, it uses information on hidden events to discover many transitions of the LCP that will not be taken. It eliminates these transitions from consideration, thereby ignoring much of the LCP graph from consideration quite early in its execution.

### 4.3.2 An Illustrative Example

We illustrate PARCOMP on an example that has been constructed to involve many interesting situations (figure 13). In this figure, the names of the events and ports associated with modules A, B, and AB are 'local' to those modules. (We will indicate the renaming of connected ports and events to common names in our narration.)

#### Structural Details

Two processes A and B are connected to form a system called AB. The `Oe1` event of A is unconnected as well as hidden; hence it is effectively ignored throughout. Event `Oe` of A is connected to event `Ie` of B, and hence whenever `Ie` is offered by B and `Oe` is asserted by A, the events would synchronize. This event is also exported as event `Oer` of AB. Thus whenever `Oe` is asserted by A, event `Oer` would be seen asserted outside AB.

Process A has a data port `!do` connected to port `?di` of B. Since this connection is hidden within AB, the data assertions on `!do` will not be visible outside AB. A also has an output port `!do2` that is connected to input port `?di` of A, output port `!do` of B, and output port `!do` of AB. The effects of these connections will be discussed momentarily. B has an input port `?hid` that is connected nowhere; the effect of querying through this port will be of interest. Finally, B has an input port `?exp` that is exposed outside AB; the effect of B's query on this port will also be of interest.

#### Behavioral Details

The above structural connections show *potentials* for interaction through events and data ports. Whether these potentials are actually utilized depends upon the protocol specifications of A and B.

Figure 12 depicts the PROTOCOL sections of processes A and B. At time 0, process A is in control state 0 and has data path state `[as]`. (Data path states are always sequences of one or more items, and we write them within square brackets, to mimic the syntax used in the textual version of the HOP specification.) While in control state 0, A keeps an output event `Oe` asserted. It also asserts the data value `!do=F(as)` so long as it stays in control state 0. It moves to state 1 when time instant 1 arrives. In control state 1, it asserts a data item, and also queries port `?di` to obtain a value for a local variable `y`. The value of variable `y` represents the value on port `?di` at time 1. Process A then moves to control state 2. Further behavior of A can be similarly understood. We indicate the state 0 of A using a darker circle because it corresponds to an explicitly named process "`A[as]`" in the absproc description of A.

Let us consider B. It offers a choice between events `Ie1` and `Ie` in state 0. The former transition will be taken if event `Ie1` is asserted (from outside B). The latter transition will be taken if event `Ie` is asserted. Since the intended semantics is one of deterministic execution the situation where `Ie1` and `Ie` are both asserted is not considered (see section 3.2.1).

If `Ie` is asserted, the data query `x=?di` will be made. After this query, B goes to control state 1. From control state 1, it goes to control state 2, and its data path state changes to `[bs, x]`. State 2 of B is shown using a dark circle because it corresponds to the explicitly named process `B1[b1s,t]`. (We have defined processes B and B1 through mutual recursion.)

Note that we show the "next data path state" only if it changes. B starts from control state 2 in data path state `[b1s,t]`. This pair is bound to `[bs,x]` by virtue of the data path state change shown along the arc $1 \rightarrow 2$.

If processes A and B are coupled using the structure shown in figure 13, and allowed to run starting them both in state 0, their behavior, as seen from outside AB, will be that of process AB in figure 12. This behavior was automatically deduced using the PARCOMP procedure. The behavior of process AB is expressed through mutual recursion, by introducing a new process (say, 'AB1') corresponding to the control state 22.

### Operational Rules Invoked in Deducing Process AB

Connections between ports and events of processes A and B are modeled by naming them to common names. (In our narration below, we will perform these renamings "as and when needed" during explanation.) Processes A and B are then composed via the $\|$ operator. Thereafter certain events and ports are internalized using the 'hide' operator.

- PARCOMP can be thought of as a procedure that generates the LCP. While doing so, events and data assertions/queries are clashed.

  Consider the move of A from 0 to 1, and B from 0 to 0. We obtain the LCP edge $00 \rightarrow 10$. We find the labels on this LCP edge using the operational rule for $\|$, as follows:

  - The set of input events is $\{Ie1\} \setminus \{Oe\}$, i.e. $\{Ie1\}$.
  - The set of output events is $\{Oe\}$.
  - The set of data assertions is $\{!do = F(as)\}$.

- In a similar fashion we consider the move of B from 0 to 1, and of A from 0 to 1, and obtain the LCP edge $00 \rightarrow 11$ as well as the labels on this edge, again using the operational rule for $\|$. In this case, since ports `!do` and `?di` are connected, we will first rename them to a common name. Then, applying the rule for $\|$ will allow us to determine that variable `x` will be bound to expression `F(as)`.

- In the LCP, there are moves $00 \rightarrow 10$ and $00 \rightarrow 11$. Out of these, edge $00 \rightarrow 10$ is labeled by `Ie1`. This is an unsynchronized event. Further it is hidden. Therefore the LCP edge $00 \rightarrow 10$ is pruned.

- During the move through edge $00 \rightarrow 11$, event `Oe` is asserted. Since this is exported via `Oer`, we see `Oer` being asserted by AB during the first transition. However, port `!do` is hidden, and so we do not see this data assertion being asserted by AB. The value communication does happen, albeit internally. The effect can be seen in $x$ being bound to the expression $F(as)$ in the next data path state of process AB that is recorded along the transition $11 \rightarrow 22$.

- PARCOMP proceeds in this fashion and eventually *re-encounters* state 00. It now has to compute PARCOMP of A and B which are (respectively) in data path states `NS-A(...)` and `NS-B(...)`. However we have already computed the PARCOMP of A and B for data path states (respectively) `as` and `bs`—these are free variables, and hence more general

than `NS-A(...)` and `NS-B(...)`. Hence nothing is to be gained by doing PARCOMP again, and so the algorithm stops.

The other interesting things that happen along the way are:

- The data assertion `!dot=lub(G(x),as)` is produced by AB at time 1, as a result of the "collision" of the data assertions `!do2=as` by A and `!do=G(x)` by B.
- The assertion `!dor=J(F(as),H(lub(G(F(as)), as)))` made at time 3 is explained thus: there is an assertion made by B at time 3. This assertion is `J(t,z)`. However by now, `t` and `z` have accumulated value bindings, and these value bindings are substituted in. Thus we see that the behavior of AB represents the effects of value communications between A and B in a closed form.
- A final point of interest is the occurrence of the term `UB` in the next data path expression when going from state 44 of AB to state 00. `UB` stands for "unbound", and results from the query that B performed on its hidden port `?hid`. So long as this `UB` value is never 'used', the system can compute along safely. An example would be this: if B were an OR gate and if one of its inputs is already 1, then the other input is UB. (UB will be bound to HOP's HIZ value 'Z', or to boolean False 'F', depending on the actual IC technology used.)

# 5 Experiments with PARCOMP

In this section we present various experiments conducted using PARCOMP.

## 5.1 Introducing Protocol Errors

We deliberately introduced mistakes into the stack controller and wanted to see if PARCOMP could detect these errors. Here is a specific experiment: take the process SCTL defined in figure 7, and delete the `Oread` event that is generated after synchronizing on event `Itop`. PARCOMP is able to detect this as an error.

This is possible because of the following reason. By omitting `Oread`, the SCTL process does not generate any of the choices that MEM offers at that moment. Thus the behavior of MEM beyond this point is not defined. Hence the behavior of the stack beyond this point is not defined.

The results of PARCOMP with this erroneous SCTL are shown in figure 14. The inferred Absproc has a transition from state 000 to state STOP, which is a dead-end. A STOP control state in a process is indicative of a design error, because a hardware system's behavior must be defined for every time instant. Thus when a STOP state is generated during PARCOMP, it issues a warning to the user. This feature of PARCOMP can help ensure that timing protocols are *mutually compatible*. Much like in type-checking, the assumption is that in a majority of cases only one process would be "wrong" relative to the other; that is, we won't make "compatible mistakes" in two systems, at the same time.

However note that not all timing errors can be caught in the above manner. For example, it is possible to have an execution trace where two events $ie1$ and $ie2$ synchronize, but in the reverse order.

## 5.2 Pipelining the Stack

The inferred behavior of the Stack presented in figure 11 shows that it takes 3 ticks to complete the *push* operation. Probing the reasons for this, we see that SCTL is the source of this time wastage. It accepts `Ipush` during the first tick, does `Oup` during the second, and `Owrite` during the third; then only goes back to state 0.

We can overlap the last `Owrite` operation with the awaiting of the next command on the stack. Doing so, we would have pipelined the stack. The controller used for this purpose is PCTL, shown in figure 15. After accepting `Ipush` and performing `Oup`, PCTL goes into control state 3. Here while it awaits the next stack operation, it performs the deferred `Owrite` operation.

Using PCTL and the same old MEM and CTR, PARCOMP infers the behavior shown in figure 16. This behavioral description shows all the modes of behavior of the stack. We will study some of these modes in the next section.

## 5.3 Testing the Pipelined Stack, aided by PARCOMP

How do we know that the pipelined stack is correct? One way is to formally verify it against a requirements specification. We do not take this approach in this paper.

Let us instead test the pipelined stack, to gain some confidence in its correctness. Let us describe a *tester process* in HOP that would apply the following sequence of operations:

$$reset(stack); push(stack, 1); push(stack, 2); pop(stack); top(stack).$$

The expected result of this test is 1.

In order to test the stack, we should apply the above sequence of commands observing proper timings for command invocations, data assertions from outside, and the data query for the result of the *top* operation. It is our understanding of the timing as well as functionality of the stack that we wish to confirm through testing. The tester so constructed is shown in figure 17.

We can compose the tester and the "testee" (the pipelined stack) using PARCOMP, and thus obtain a single process that embodies all observable aspects of the collective behavior of the tester+testee. We can then run this single resultant process. The resultant process is shown in figure 18. This approach has many practical advantages, and they are discussed in the following subsections.

### 5.3.1 Detecting Timing Errors in Tester Processes Statically

PARCOMP can reveal certain timing errors in the tester, relative to the testee. In these cases, wasteful simulation needn't be performed, and instead the error can be corrected.

### 5.3.2 Obtaining Symbolic Simulation Results Without Simulation

As figure 18 shows, the inferred process reveals (approximately) how the simulation would proceed. For instance, it tells us that the final result delivered by the *top* operation is:

<< Figure on separate sheet >>

Figure 12: Processes A, B, and AB

<< Figure on separate sheet >>

Figure 13: The Realization of the System AB

<< Figure on separate sheet >>

Figure 14: Inferred Behavior of the Stack using an Erroneous SCTL

<< Figure on separate sheet >>

Figure 15: The Pipelined Stack Controller

<< Figure on separate sheet >>

Figure 16: Inferred Behavior of the Pipelined Stack (one that uses PCTL)

<< Figure on separate sheet >>

Figure 17: A Tester Process for the Pipelined Stack

<< Figure on separate sheet >>

Figure 18: Composition of the Tester and the Testee (the pipelined Stack)

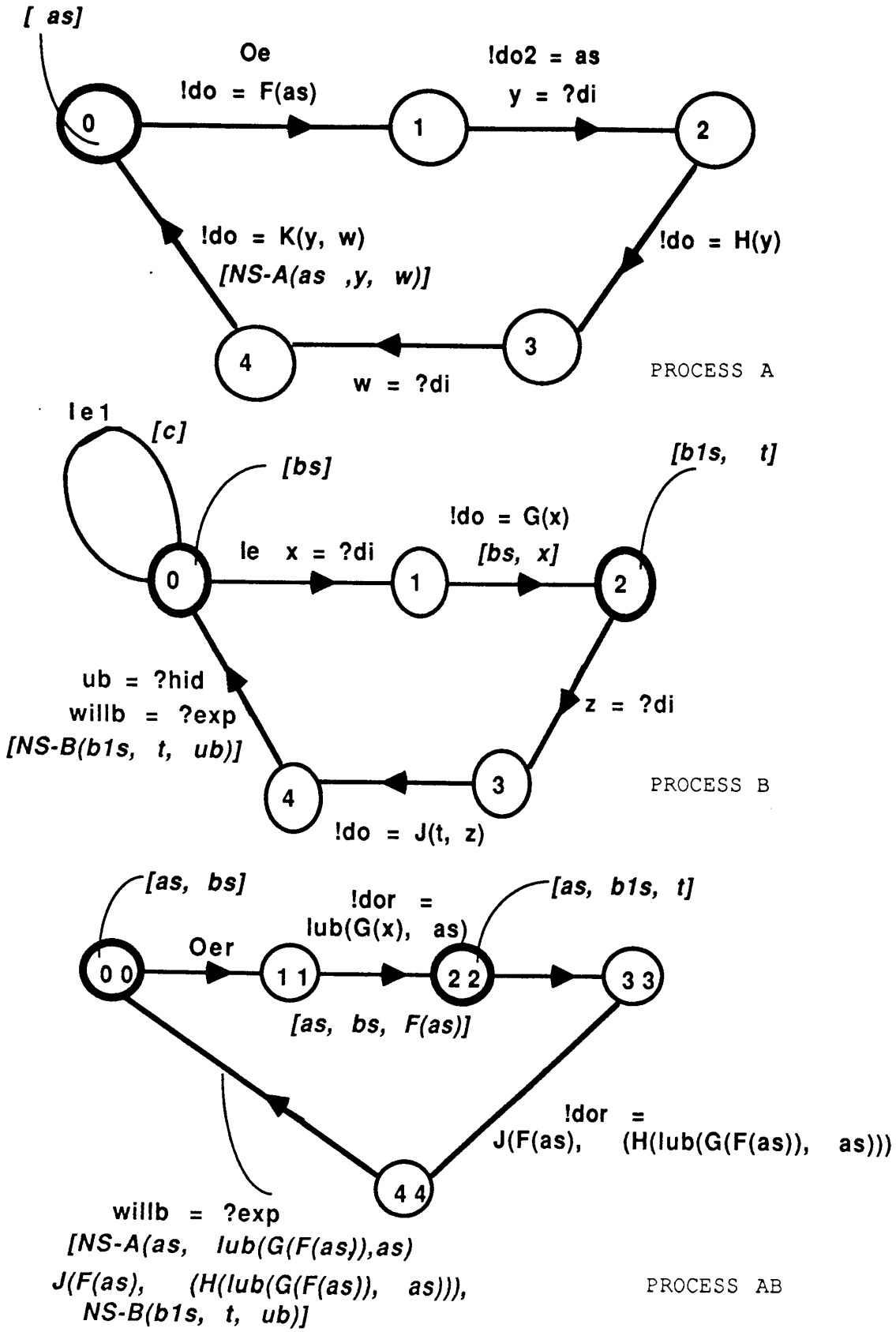

[ as]
Oe
!do = F(as)
!do2 = as
y = ?di
0
1
2
!do = K(y, w)
[NS-A(as ,y, w)]
!do = H(y)
4
3
w = ?di
PROCESS A
le1
[c]
[bs]
[b1s, t]
!do = G(x)
le x = ?di
[bs, x]
0
1
2
ub = ?hid
willb = ?exp
[NS-B(b1s, t, ub)]
z = ?di
4
3
!do = J(t, z)
PROCESS B
[as, bs]
!dor =
lub(G(x), as)
[as, b1s, t]
Oer
0 0
1 1
2 2
3 3
[as, bs, F(as)]
!dor =
J(F(as), (H(lub(G(F(as)), as)))
4 4
willb = ?exp
[NS-A(as, lub(G(F(as)),as)
J(F(as), (H(lub(G(F(as)), as))),
NS-B(b1s, t, ub)]
PROCESS AB

Figure 12

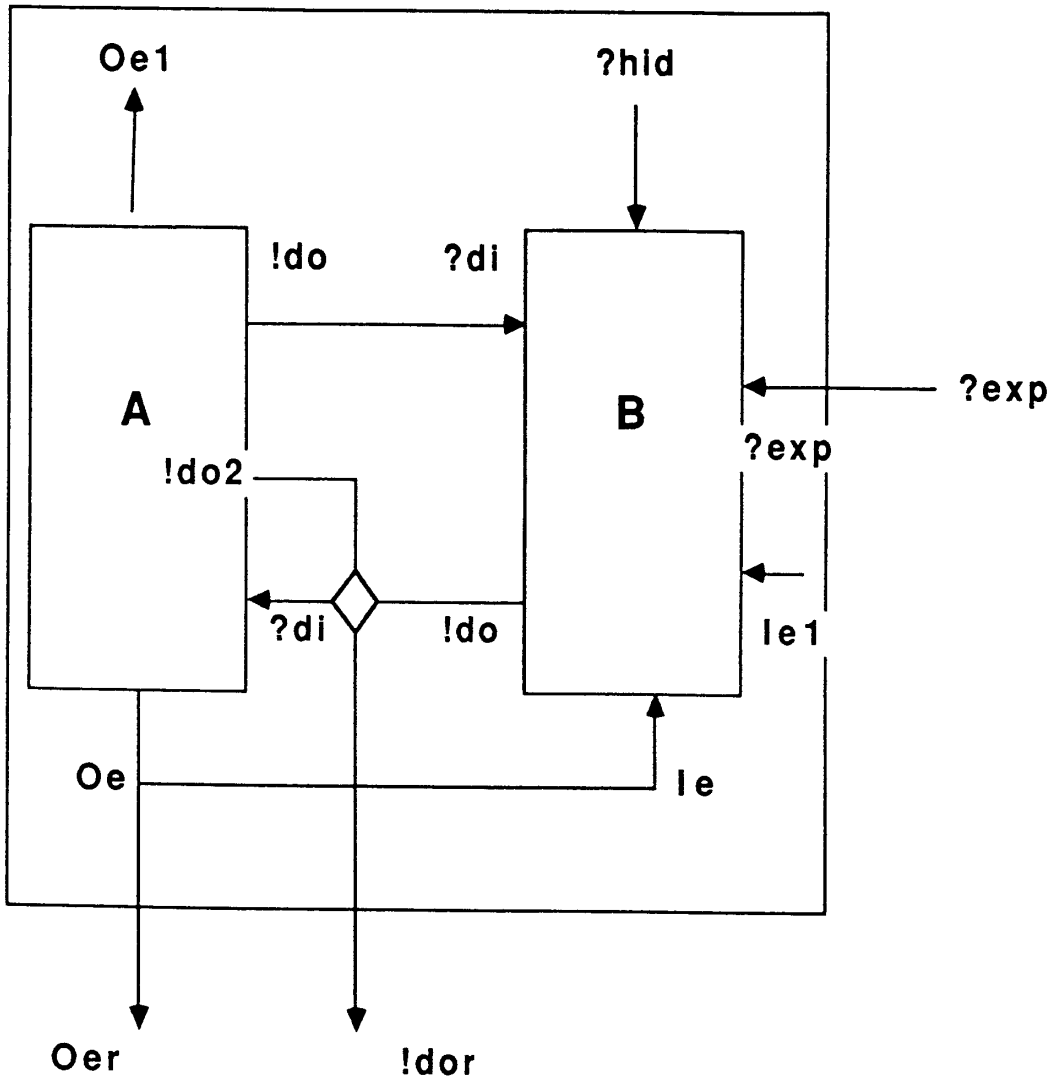



Figure 13

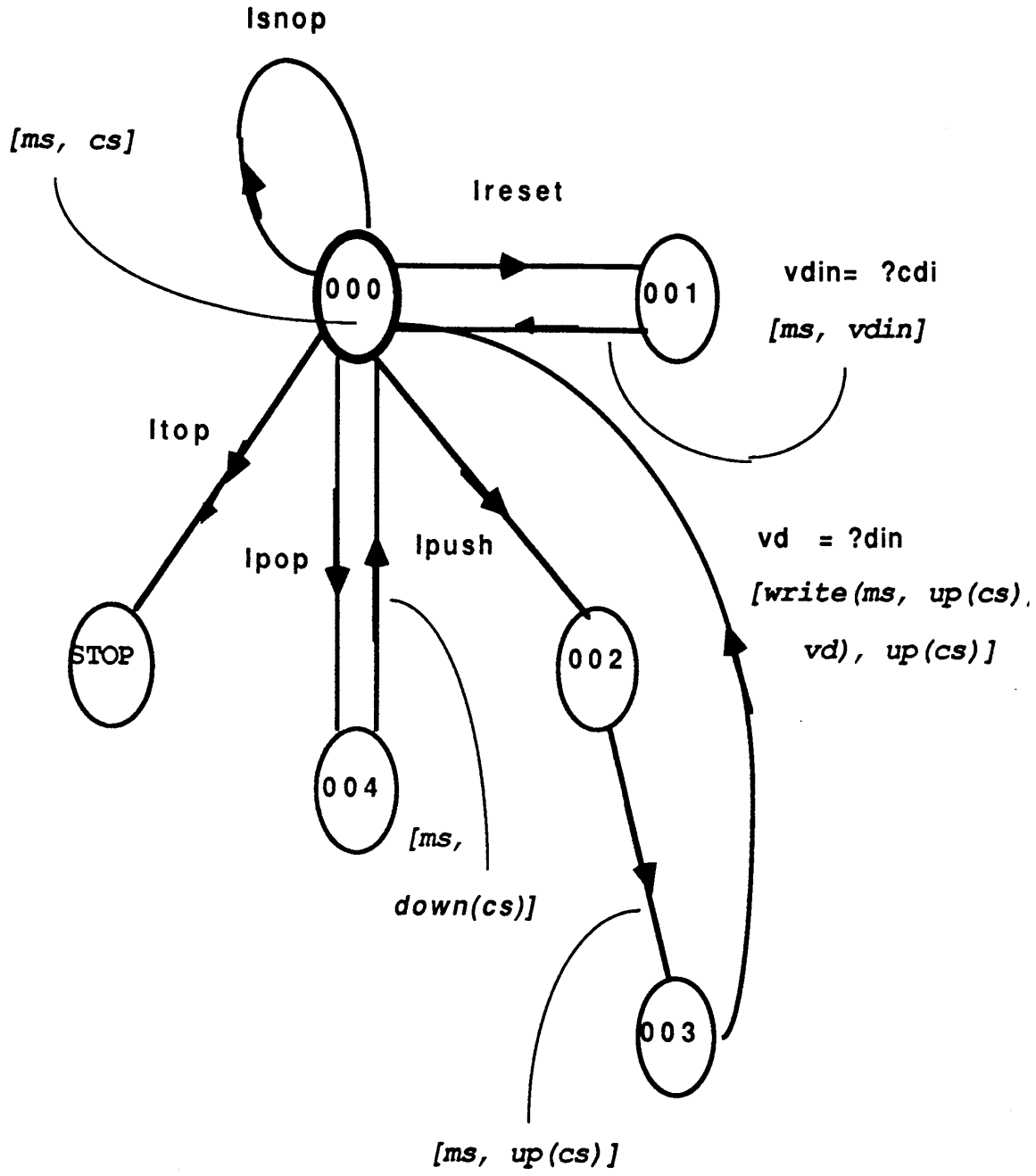

lsnop
[ms, cs]
lreset
000
001
vdin= ?cdi
[ms, vdin]
ltop
lpop
lpush
vd = ?din
[write(ms, up(cs),
vd), up(cs)]
STOP
002
004
[ms,
down(cs)]
003
[ms, up(cs)]

Figure 14

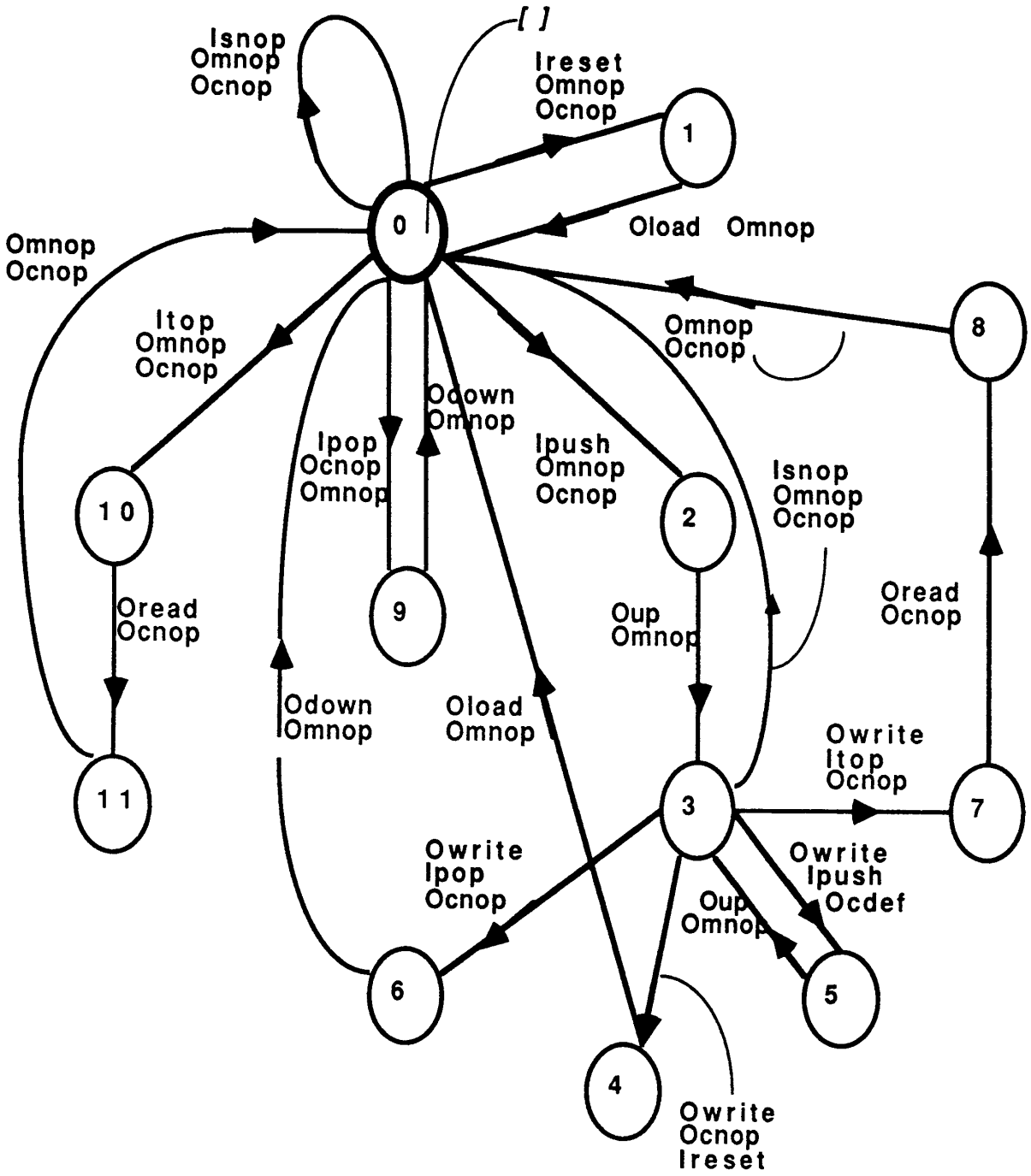

Isnop
Omnop
Ocnop
[ ]
Ireset
Omnop
Ocnop
1
0
Oload Omnop
Omnop
Ocnop
Itop
Omnop
Ocnop
Odown
Omnop
Ipop
Ocnop
Omnop
Ipush
Omnop
Ocnop
Omnop
Ocnop
8
10
2
Isnop
Omnop
Ocnop
Oread
Ocnop
9
Oup
Omnop
Oread
Ocnop
Odown
Omnop
Oload
Omnop
Owrite
Itop
Ocnop
11
3
7
Owrite
Ipop
Ocnop
Owrite
Ipush
Ocdef
Oup
Omnop
6
5
4
Owrite
Ocnop
Ireset

Figure 15

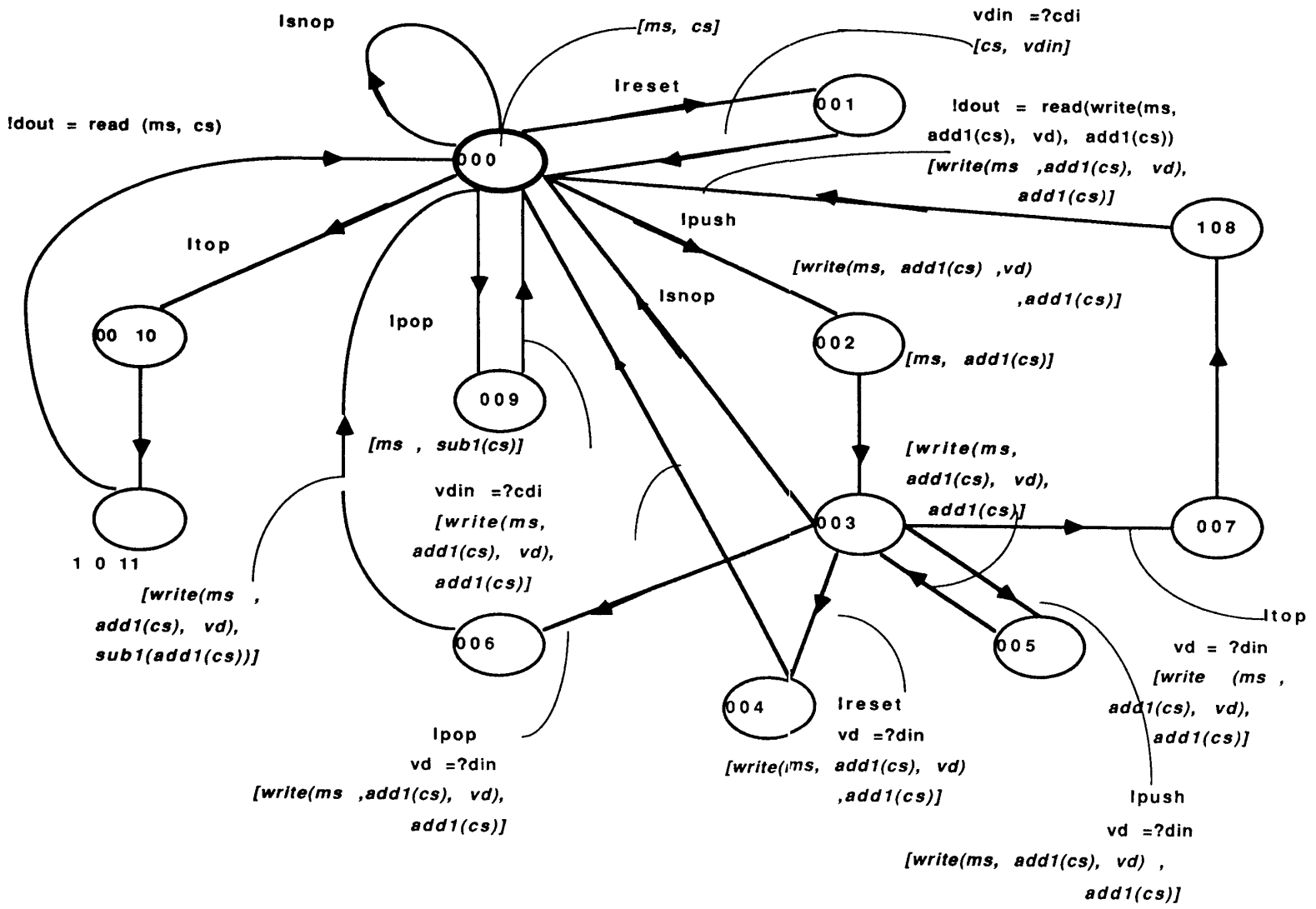

!snop
[ms, cs]
vdin =?cdi
[cs, vdin]
!reset
001
!dout = read(write(ms,
add1(cs), vd), add1(cs))
[write(ms ,add1(cs), vd),
add1(cs)]
!dout = read (ms, cs)
000
108
!push
[write(ms, add1(cs) ,vd)
,add1(cs)]
!top
00 10
!snop
!pop
002
[ms, add1(cs)]
009
[ms , sub1(cs)]
[write(ms,
add1(cs), vd),
add1(cs)]
vdin =?cdi
[write(ms,
add1(cs), vd),
add1(cs)]
003
007
1 0 11
[write(ms ,
add1(cs), vd),
sub1(add1(cs))]
!top
vd = ?din
[write (ms ,
add1(cs), vd),
add1(cs)]
006
005
004
!pop
vd =?din
[write(ms ,add1(cs), vd),
add1(cs)]
!reset
vd =?din
[write(ms, add1(cs), vd)
,add1(cs)]
!push
vd =?din
[write(ms, add1(cs), vd) ,
add1(cs)]

Figure 16

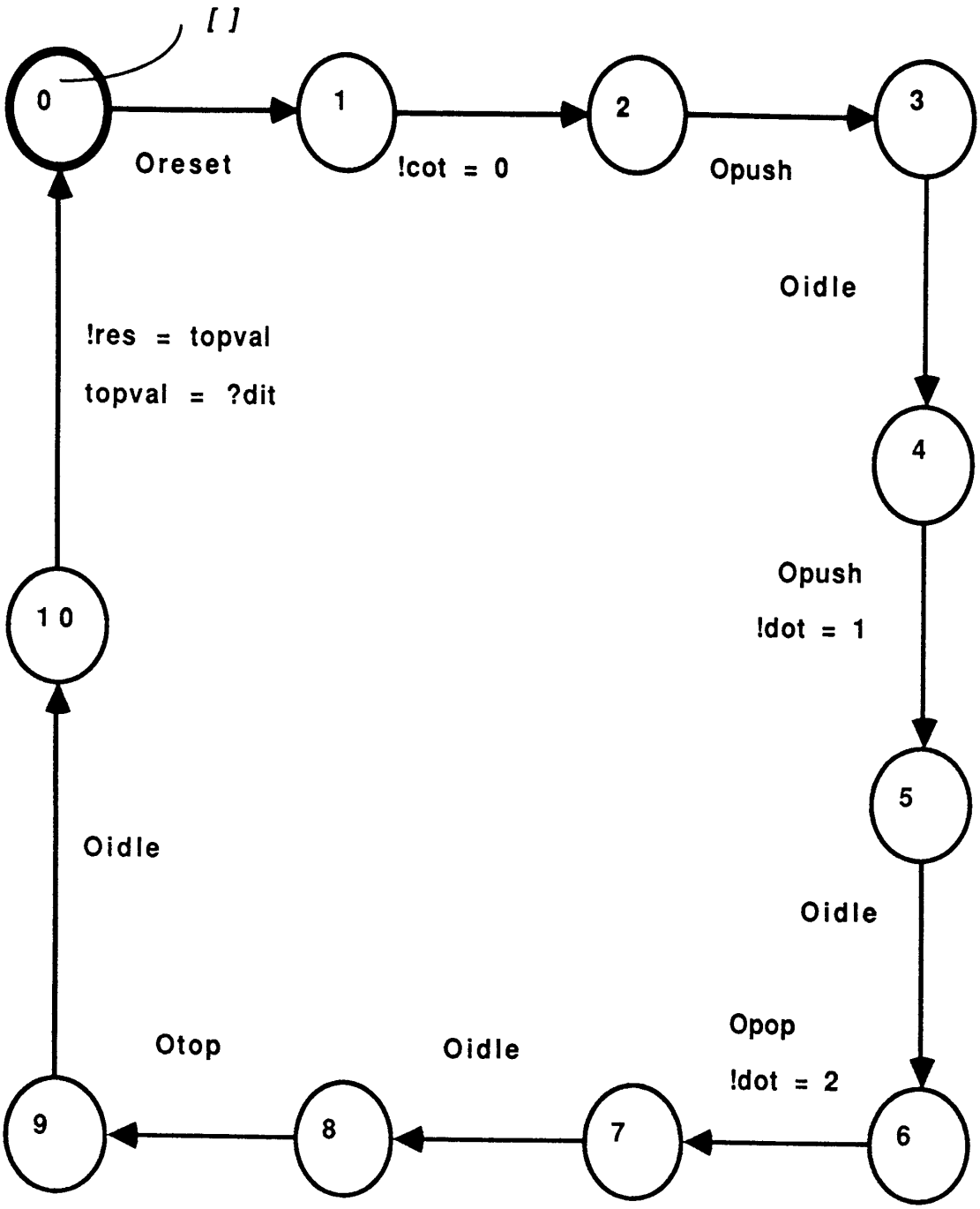

[ ]
0
1
2
3
Oreset
!cot = 0
Opush
Oidle
!res = topval
topval = ?dit
4
Opush
!dot = 1
10
5
Oidle
Oidle
Opop
!dot = 2
Otop
Oidle
9
8
7
6

Figure 17

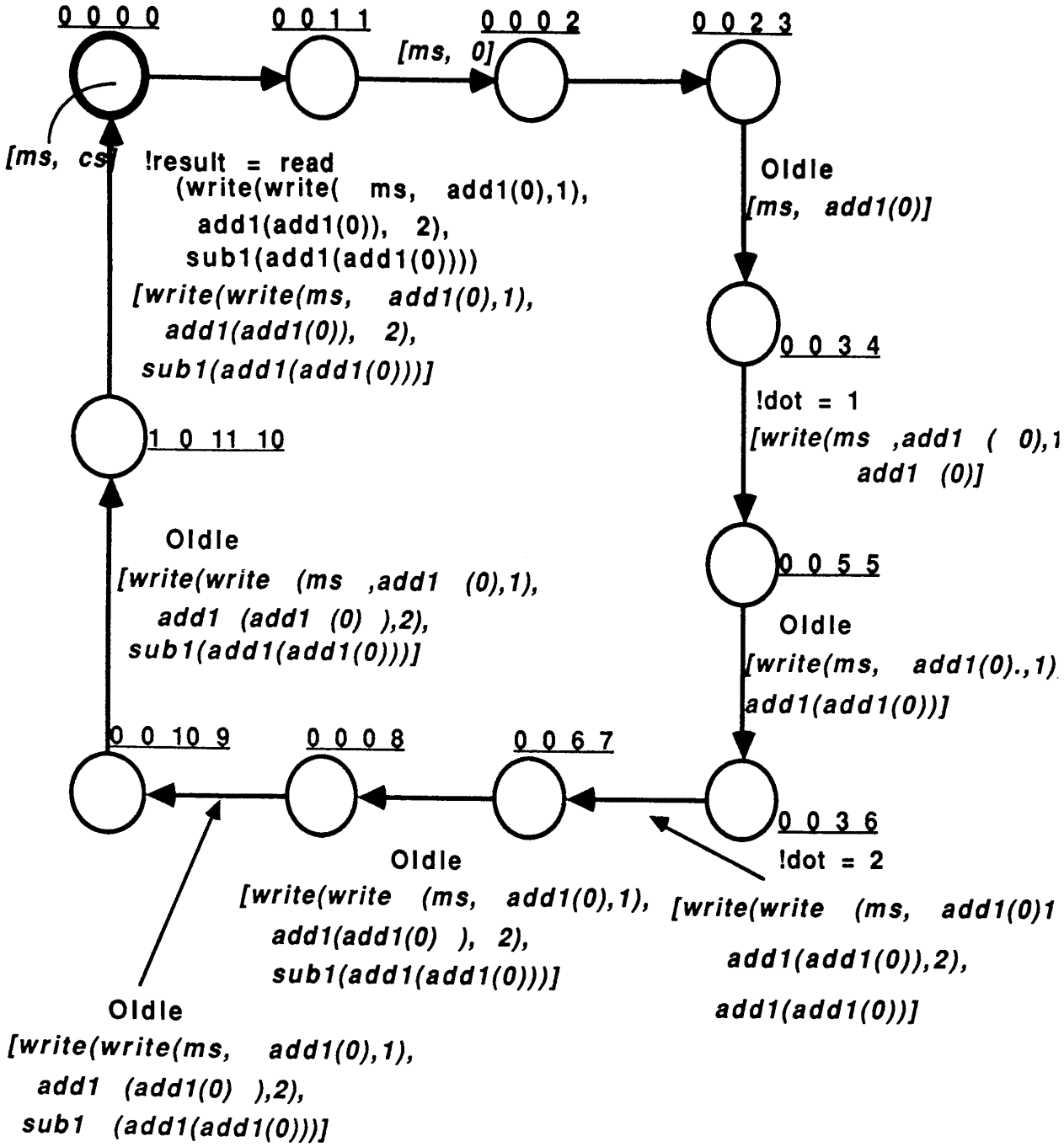

0 0 0 0
0 0 1 1
[ms, 0]
0 0 0 2
0 0 2 3
[ms, cs]
!result = read
(write(write( ms, add1(0),1),
add1(add1(0)), 2),
sub1(add1(add1(0))))
[write(write(ms, add1(0),1),
add1(add1(0)), 2),
sub1(add1(add1(0)))]
Oldle
[ms, add1(0)]
0 0 3 4
!dot = 1
[write(ms ,add1 ( 0),1
add1 (0)]
1 0 11 10
Oldle
[write(write (ms ,add1 (0),1),
add1 (add1 (0) ),2),
sub1(add1(add1(0))))]
0 0 5 5
Oldle
[write(ms, add1(0).,1)
add1(add1(0))]
0 0 10 9
0 0 0 8
0 0 6 7
0 0 3 6
!dot = 2
Oldle
[write(write (ms, add1(0),1),
add1(add1(0) ), 2),
sub1(add1(add1(0)))]
[write(write (ms, add1(0)1
add1(add1(0)),2),
add1(add1(0))]
Oldle
[write(write(ms, add1(0),1),
add1 (add1(0) ),2),
sub1 (add1(add1(0)))]

Figure 18

```
!result =
 (READ
  (WRITE (WRITE MS (ADD1 0) 1) (ADD1 (ADD1 0)) 2)
  (SUB1 (ADD1 (ADD1 0)))
 )
```

In this simple example, we can readily tell that this answer is correct; for, we can apply simple algebraic rules of ADD1 and SUB1, to simplify this data assertion to:

```
!result =
 (READ (WRITE (WRITE MS 1 1) 2 2) 1)
```

This can further be simplified to 1, using the following algebraic axiom of ordinary read-write memories:

$$read(write(m, a, d), a) = d.$$

And 1 was indeed our expected answer.

This opens up the following attractive path towards speeding up functional simulation:

1. Build an algebraic expression simplifier as a part of the abstract data type library.
2. Obtain the "tester+testee" process thru PARCOMP.
3. Extract all the the *next data-path state* and *data assertion* expressions present in this tester+testee. Simplify them using the expression simplifier.
4. Plug these simplified expressions back into the tester+testee.
5. Run detailed functional simulation on this simplified tester+testee.

### 5.3.3 Building Partial Testers

Suppose we want to supply certain test stimuli "automatically" from the tester process and some other test stimuli interactively from the keyboard. This can be very easily done in our present approach. For example, let us assume that the user wants to have control over the first data item being pushed on the stack. He/she would simply leave out the data assertion `!dot=1` from figure 17. Running PARCOMP on this "tester+testee" would result in an "unsatisfied but un-hidden" data query at time 4. When we run the HOP simulator on such an absproc, the unsatisfied data query is turned into a query from the keyboard.

Thus users may selectively add or take away events and data assertions from the tester process. Thus, a range of testers are possible. At one extreme, the tester does every data assertion and query, and so the simulation will run on its own, without user intervention. At the other extreme, the tester would do *nothing*, and the simulator would interrogate the user for every event and data input. This was a pleasant and serendipitous discovery.

### 5.3.4 Interpreted Realproc Simulator

Sometimes it may be felt necessary to simulate a collection of processes without doing PARCOMP. This need can arise, for example, during the very early stages of a design where (i) users may want to simulate a proper subset of the subprocesses; (ii) users may want to get detailed information about the innards of a system. To support this need, we have developed a run-time version of PARCOMP that is embodied in an *Realproc Interpreted Process Simulator* (RIPS).

### 5.3.5 The use of Probe Processes

Logic state analyzers are widely used to debug digital systems. In HOP, we can simulate logic state analyzers. by constructing *probe* processes.

A probe process is constructed by specifying along its transitions a *trace* of the sequence of events and data assertions of interest. Such a trace is similar to a "trigger" specification of a logic state analyzer. We can then PARCOMP the probe process with the submodules of a system, and then simulate the system.

Here is a probe process that can be used with the pipelined stack:

```
PROBE <= Iwrite ~> Iwrite ~> Iwrite ~> Iread -> !probeout = ''Success''
```

The operator `~>` is an abbreviation for "busy wait until the following input event". This derived operator is available in HOP, and can be expressed in terms of `->`.

If this probe process were to be composed with the pipelined stack and tested using figure 17, it will sense whether the memory is being subject to three writes and one read. If so it will print `''Success''` on the `!probeout` port. For the command sequence *push*; *push*; *pop*; *top* applied by our tester, this trace must manifest on the memory subprocess. Probe processes may, after sensing the trigger condition, start acquiring data, and may even act like tester processes by supplying test patterns.

### 5.3.6 Checking for Representation Invariants

Probe processes may be used for flagging the violation of of *representation invariants* during the course of operation of a module. Representation invariants [25] are predicates that describe the consistent internal states of a module. As an example, consider a simple associative memory (AM) with 4 locations. A representation invariant found in most AMs is: "AM never contains duplicate entries". Stated formally,

$$\forall x \; unary(assoc\_srch(AM, x)).$$

This says that $d$, the result of doing an associative search, is always a unary quantity. If the unary pattern is "0000", it indicates that the search "missed". If the pattern is "0010", it indicates that there was a hit at location 3. If pattern is "0101", it indicates that $x$ was found in location 0 and 3; this is erroneous. A probe process to detect this condition is:

```
NODUP <= Isearch, x=?srchdata -> if(unary(x), NODUP, ERROR)
ERROR <= !probeout = ''Error'' -> STOP
```

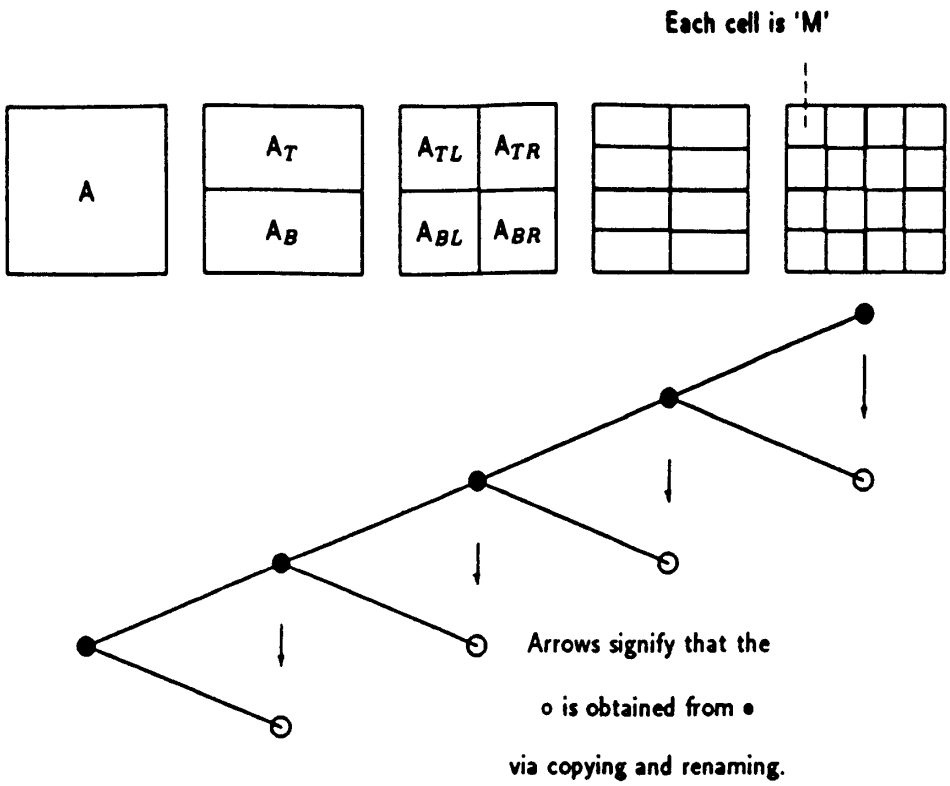



Figure 19: Divide and Conquer PARCOMP

The probe process NODUP samples the Isearch event that triggers the associative search. It samples the search's result, x, also. Then if x is found to be unary, it goes back to behave like NODUP. Else it behaves like the ERROR process.

This technique has one limitation: quite often, the entire internal state of a module is not observable through its output ports. To overcome this limitation, we are investigating the use of *daemons*—data driven procedures—that can directly monitor the ADT object states.

---

# 6 A Divide-and-conquer PARCOMP, 'PCDC'

This section sketches a variant of PARCOMP that exploits the high degree of regularity found in many real-world VLSI arrays. The arrays on which PCDC can be applied need not be regular in their computation and communication (as systolic arrays are); they need only be regular in their *structure*. To clarify this point, consider a 4x4 array of cells. Let each cell support 4 different operations. Suppose that for certain data inputs some cells are doing operation #1 while others are doing operation #2, etc. PCDC can be applied to this array of cells. In addition, PCDC can be applied to arrays in which there exist embedded global busses—something that systolic systems do not have.

For the same MxN array PCDC is expected to run faster than PARCOMP. To understand how this is achieved, consider the array $A$ shown in figure 19. 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. Now consider the problem of computing $PARCOMP(A)$; *i.e.* the composition of all the $M$s constituting $A$.

A simple but crucial property enjoyed by $PARCOMP$ is that is both commutative and associative. This is exploited by PCDC. It splits $A$ into two halves, say $A_T$ standing for "the top of $A$" and $A_B$, standing for "the bottom of $A$", and uses the property:

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

Since $A_T$ and $A_B$ differ only in the names of their external ports, PCDC need compute *only* $PARCOMP(A_T)$. $PARCOMP(A_B)$ can be obtained from this, by renaming the ports of $A_T$ to the corresponding ports of $A_B$. This division process can be carried down to the leaf cells, as depicted in figure 19.

PCDC has been implemented and applied to a few real-world arrays. For a large class of practically occurring arrays, PCDC's run time is $O(N^2 log(N))$ where $N$ is the number of cells in the regular array. PCDC's complexity analysis and experimental results will be reported in [16].

# 7 Summary of the Paper

We presented a language "Hardware viewed as Objects and Processes" (HOP) for specifying the structure, behavior, and timing of hardware systems. HOP embodies a simple process model for lock-step synchronous processes.

We presented the communication primitives of HOP, illustrated HOP through several examples, and then presented its operational semantics. Several design automation algorithms—especially PARCOMP—were then examined in detail. The results presented herein were obtained from our implementation of the HOP design system. Section A.1 presents an overview of this system. It has a working prototype, currently written in Common Lisp and FROBS [34].

Though we have taken simple examples in this paper, we have worked out some larger examples as well. A few large as well as intricate systems specified to date are: (i) A Huffman encoder; (ii) A cache memory system; (iii) A major portion of the Texas Instruments Micro Sequencer chip 74AS890 [1]. We have run the first two of these examples through PARCOMP as well as simulated the inferred process. These experiments have confirmed the utility of most of our design decisions, some important ones being:

- separation of data I/O from control I/O;
- separation of data state from control state;
- the PARCOMP algorithm, and its applications for deducing succinct behavioral descriptions from structural descriptions;
- capitalizing on the locality of hidden events for detecting sequencing errors as well as reducing the run-time of PARCOMP.

Work is currently in progress in generalizing HOP to encompass 'truly concurrent' systems.

# References


[1] Venkatesh Akella. A micro assembler for the ti 74as890 micro sequencer. Technical Report UUCS-88-016, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1988.

[2] T.S. Anantharaman, E.M. Clarke, M.J. Foster, and B. Mishra. Compiling path expressions into vlsi circuits. In *Proceedings of the 12th Symposium on Principles of Programming Languages.* ACM, January 1985.

[3] Mario R. Barbacci. Instruction set processor specifications (isps): The notation and its applications. *IEEE Transactions on Computers*, C-30(1):24–40, January 1981.

[4] 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.

[5] M. Browne, Edmund Clarke, D. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. In *Proceedings of the Seventh International Conference on Computer Hardware Description Languages*, pages 98–113. North-Holland, 1985.

[6] Randall E. Bryant. A switch level model and simulator for MOS digital systems. *IEEE Transactions on Computer*, C-33:160–177, February 1984.

[7] Albert Camilleri, Michael C. Gordon, and Tom Melham. Hardware specification and verification using higher order logic. In *Processings of the IFIP WG 10.2 Working Conference on "From HDL Descriptions to Guaranteed Correct Circuit Designs", Grenoble, August 1986.* North-Holland, 1986.

[8] Vincenza Carchiolo, Alberto Faro, Orazio Mirabella, Giuseppe Pappalardo, and Giuseppe Scollo. A LOTOS specification of the PROWAY highway service. *IEEE Transactions on Computers*, C-35(11):949–968, November 1986.

[9] 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.

[10] Tam-Anh Chu. Synthesis of self-timed vlsi circuits from graph-theoretic specifications. In *International Workshop on Petri Nets and Performance Models, Madison, Wisconsin*, August 1987. See also MIT VLSI Memo no.87-410, September 1987, with the same title.

[11] Avra Cohn. Correctness properties of the Viper block model: The second level. In *1988 Banff Workshop on Hardware Verification.* Springer Verlag, 1988.

[12] Stephen Garland, John Guttag, and Jorgen Staunstrup. Verification of vlsi circuits using lp. In George Milne, editor, *1988 Glasgow Workshop (IFIP WG 10.2) on Hardware Verification*, 1988.

[13] Ganesh C. Gopalakrishnan. *From Algebraic Specifications to Correct VLSI Systems.* PhD thesis, Dept. of Computer Science, State University of New York, December 1986. (Also Tech. Report UU-CS-86-117 of Univ. of Utah).

[14] Ganesh C. Gopalakrishnan. Synthesizing synchronous digital vlsi controllers using petri nets. In *International Workshop on Petri Nets and Performance Models, Madison, Wisconsin*, August 1987.

[15] Ganesh C. Gopalakrishnan, Richard M. Fujimoto, Venkatesh Akella, N.S. Mani, and Kevin N. Smith. Specification-driven design of custom architectures in hop. In P.A.Subrahmanyam and G.Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, chapter 3, pages 128–170. Springer-Verlag, 1989.

[16] Ganesh C. Gopalakrishnan, Narayana S. Mani, and Venkatesh Akella. A tool for the parallel composition of finite-state processes with applications to hardware validation. In *Workshop on Automatic Verification Methods for Finite-State Systems, Grenoble, France, June 12-14, 1989.* Springer Verlag, 1989. *Accepted for Publication.*

[17] Ganesh C. Gopalakrishnan and Mandayam K. Srivas. Implementing functional programs using mutable abstract data types. *Information Processing Letters*, 26(6):277–286, January 1988.

[18] Ganesh C. Gopalakrishnan, Mandayam K. Srivas, and David R. Smith. From algebraic specifications to correct vlsi circuits. In D.Borrione, editor, *From HDL Descriptions to Guaranted Correct Circuit Designs*, pages 197–225. North-Holland, 1987. (Proc of the IFIP WG 10.2 Working Conference with the same title.).

[19] Richard H. Lathrop Robert J. Hall and Robert S. Kirk. Functional abstraction from structure in vlsi simulation models. In *Proc. 24st Design Automation Conference*, pages 822–828, 1987.

[20] Matthew Hennessy. Proving systolic systems correct. Technical Report CSR-162-84, Department of Computer Science, University of Edinburg, June 1984.

[21] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985. *Definitive discussion of CSP, circa 1985.*

[22] Stephen Johnson, B. Bose, and C. Boyer. A tactical framework for hardware design. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

[23] Jeffrey Joyce and Graham Birtwistle. Proving a computer correct in higher order logic. Technical Report 85/208/21, Dept. of Computer Science, Univ. of Calgary, August 1985.

[24] Robert R. Kessler, Eric G. Muehle, and Jed Krohnfeldt. Efficient structures for knowledge-based applications. In *Proc. of the 1987 Rocky Mountain AI Conference*, June 1987.

[25] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, 1986. ISBN-0-07-037996-3.

[26] L. Logrippo, A. Obaid, J.P.Briand, and M.C. Fehri. An interpreter for LOTOS, a specification language for distributed systems. *Software—Practice and Experience*, 18(4):365–385, April 1988.

[27] Zohar Manna. *Mathematical Theory of Computation.* New York: McGraw-Hill, 1974.

[28] John Merk, John Lalonde, and Ganesh Gopalakrishnan. Adtp user's manual. Requirements Specification and User Manual for the Abstract Data Type definition Package (ADTP), Software Engineering Lab., Spring 1988.

[29] 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.

[30] George J. Milne. Circal: A calculus for circuit description. *Integration*, (1):121–160, 1983.

[31] 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.

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

[33] S. Morpurgo, A. Hunger, M. Melgara, and C. Segre. Rtl test generation and validation for vlsi: An integrated set of tools for karl. In *Proc. Seventh International Symposium on Computer Hardware Description Languages*, pages 261–271. North Holland, 1985.

[34] Eric G. Muehle. Frobs: A merger of two knowledge representation paradigms. Master's thesis, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, December 1987. FROBS Stands for Frames+Objects.

[35] P. Narendran and J. Stillman. Hardware verification in the interactive vhdl workstation. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 235–255. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

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

[37] R.C.Sekar and Mandayam Srivas. Formal verification of a microprocessor using equational techniques. In G.Birtwistle and P.A.Subrahmanyam, editors, *1988 Banff Hardware Verification Workshop, Banff, June 1988*, 1988. Invited Paper, to appear as a chapter in a forthcoming Springer-Verlag book.

[38] Mary Sheeran. Design of regular hardware structures using higher order functions. In *Proceedings of the Functional Programming and Computer Architecture Conference.* Springer-Verlag, LNCS 201, September 1985. Nancy, France.

[39] Jan Snepscheut. *Trace Theory and VLSI Design.* Springer Verlag, 1985. LNCS 200.

[40] Pashupathy A. Subramaniam. Overview of a conceptual and formal basis for an automatable high level design paradigm for integrated systems. In *Proceedings of the International Conference for Computer Design and VLSI, Westchester*, pages 647–651, 1983.

[41] Vhdl language reference manual, August 1985. *Intermetrics Report IR-MD-045-2; See also IEEE Design and Test, April 1986.*

[42] W.F.Clocksin. Logic programming and digital circuit analysis. *Journal of Logic Programming*, (4):59–82, 1987.

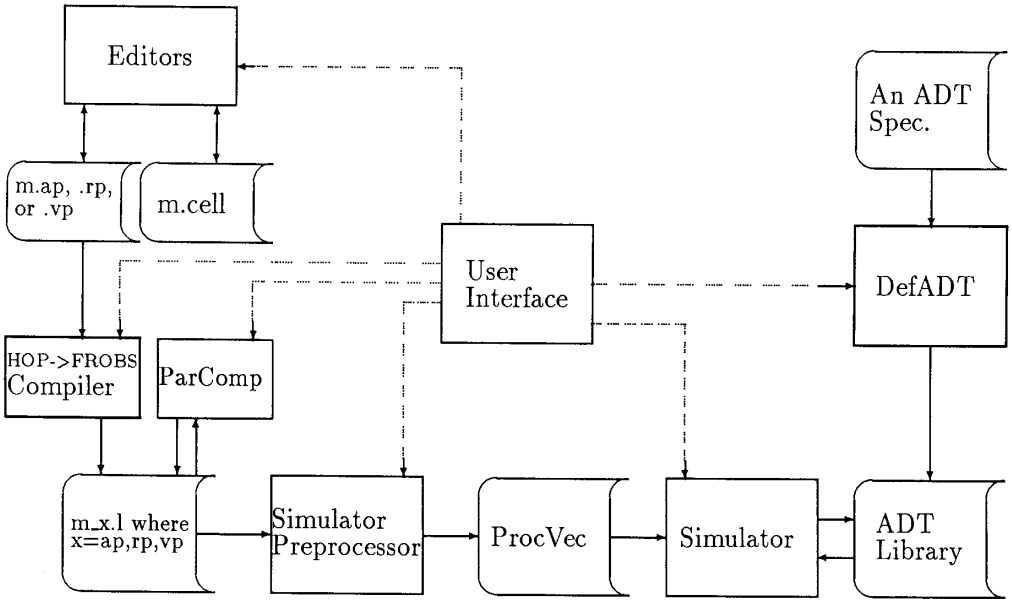



Figure 20: Data Flow Diagram of the HOP Design System

# A Appendix

## A.1 A Brief Description of the HOP Design System

Figure 20 illustrates the data flow diagram of the HOP design system. The rectangular boxes indicate functional units, and boxes with curved sides indicate intermediate storage units. Dotted lines show the flow of control, and solid lines show the flow of data. Currently, working prototypes exist for all the functional units shown in this figure.

Input specifications are entered through text editors. File name extensions `.ap`, `.rp`, and `.vp` refer to absproc, realproc, and vecproc. Cell specifications are entered using an available VLSI design system. HOP specifications are compiled into FROBS representations using the HOP→FROBS compiler. The algorithm PARCOMP can now be applied on Realprocs and Vecprocs (presently implemented only for Realprocs). PARCOMP infers functionally equivalent absproc specifications from Realproc and Vecproc specifications.

The simulator preprocessor compiles the FROBS database into a form suitable for the simulator (under development). A data type definition mechanism has been implemented using FROBS [28]. During simulation, the simulator will be called upon to evaluate functional expressions that compute new datapath states as well as output port values. These will be achieved by invoking the operations defined on the various data types. FROBS supports

daemons that can help probe simulation results, as explained in section 5.3.6.