HOP : A Process Model for Synchronous Hardware Semantics, and Experiments in Process Composition

Ganesh C. Gopalakrishnan Richard M. Fujimoto
Venkatesh Akella and Narayana S. Mani

UUCS-88-012
August 15, 1988

HOP: A Process Model for Synchronous Hardware Semantics, and Experiments in Process Composition

Ganesh C. Gopalakrishnan[†], *Richard M. Fujimoto*[††]
Venkatesh Akella and Narayana S. Mani
Dept. of Computer Science, University of Utah
Salt Lake City, Utah 84112, U.S.A

Abstract. *We present 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. An* absproc *specification written in HOP describes the externally observable behavior of a process. A collection of* absprocs *may be composed to form a larger process, using the operators* parallel composition, renaming, *and* hiding.

In this paper we present the communication primitives of HOP, illustrate HOP through several examples, and then present its operational semantics. Then we present the role played by HOP in in three VLSI design activities: (i) inferring concise behavioral descriptions of systems from their structural descriptions; (ii) static detection of control timing errors during behavioral inferrence; (iii) productive and runtime efficient functional simulation using the inferred behavior.

†Supported in part by the National Science Foundation via MIP-8710874
††Supported in part by ONR Contract Number 00014-87K-0184

Note: Some portions of this paper will appear in the Proceedings of the 1988 Banff Workshop on Hardware Verification, Banff, Canada, June 1988 (to be published by Springer-Verlag).

Contents

List of Figures

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 [21,35] Prolog [38], Petri Nets [7], Temporal Logic [4], various Calculii of Communicating Systems [26,16] Trace Theory [36], Higher Order Logic [6,22], Algebraic Specifications and Equational Techniques [14,37,32], Synchronized Transition Systems [9], and Path Expressions [1], 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 writing formal specifications—especially for unambiguous documentation of designs, supporting design automation activities, etc.—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 runtime efficient functional simulation using the inferred behavior. The main contributions of this work are believed to be: (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 on a working implementation of HOP and PARCOMP.

Despite being a formal specification language, HOP specifications are easy to understand. HOP can intuitively model the intricate timing protocols that synchronous hardware systems exhibit. It can model commonly used structures in VLSI, such as through connections and regular arrays, directly. 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 our specific results to date.

Motivation

Specifying the timing protocols and the functional behavior of synchronous systems with clarity is quite important. More importantly, the functional details are also intricately interwoven with timing. *The examples in this paper are chosen to illustrate the clarity with which HOP can specify such intricate behaviors.*

It has been reported that the complete formal verification of even extremely simple ICs is at present a challenging task [8]. More importantly, impressive results with theorem provers have almost always been exhibited by persons who played a major role in developing the theorem prover (and hence knew its innards)—not by end-users of theorem provers. Until these situations change significantly, the main uses of formal specifications will be for its *indirect benefits*—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 presents the HOP language, and illustrates the various language concepts introduced through the example of a simple stack.

- Section 3 presents the operational semantics of HOP. (Note: If the reader were to feel intimidated by the formal notation in this section, then he/she may read this section cursorily without much loss.)

- Section 4 illustrates PARCOMP on a simple example, and also illustrates how each rule of the the operational semantics are 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 (*often*) 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 presents a divide-and-conquer version of PARCOMP. This technique exploits two disparate facts: (i) that the PARCOMP operator is commutative and associative; (ii) that VLSI systems have a high *replication factor*—i.e. the ratio of the total number of modules to the number of *different* modules.

- Section 7 presents our conclusions; In appendix A.2, we briefly describe the HOP design system that was used to produce the results reported.

1.1 Understanding the Modeling Philosophy of HOP

One significant aspect of HOP is that it emphasizes the use of abstract data types for hardware modeling. This was motivated by the positive results from the first author's past work with the SBL language [14,10,11]. We now present through a simple example the essence of HOP.

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*) would be provided via functional expressions that use operators on the stack pointer and memory types. The stack state would be modeled as a tuple $< ctr, mem >$, consisting of the counter and memory states. The operation *push* can be modeled via the functional expression:

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

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 [21].

As we showed in our past work with SBL, these kinds of specifications may be implemented in hardware by synthesizing controller modules that "fire" the operations *write*, *read*, *add*1, etc. in an applicative order (actually the *in situ evaluation order* [13], 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 [10,34].

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 a more general notion than that of an operation. It is like a *trace* of [17]. A mode of behavior is best characterized as a *finitely describable* (and often finite) sequence of *events*, *data input actions*, and *data output actions*.

For example, consider a memory data type that has a *read* operation. A realization of the memory has *many possible* (depending on design decisions such as pipelining etc.) *read* modes of behaviors. One such mode of behavior consist of a *read* trigger 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 ith read event trigger and ith address input must precede the ith 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 upto (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. The functions and their inputs and outputs must be inter-woven with the timing aspects of the mode of behavior.

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 to a person is by drawing the picture of a Deterministic Finite-state Automaton (DFA). One may ask, "why not use DFAs directly for specifying hardware"?

This question is being considered mainly for two reasons. For one, in this paper we portray HOP process specifications through "DFA-like graphs", and we want to avoid the readers trivializing HOP as a DFA specification language. For another, it is widely known from human studies 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 related aspects are captured by *annotating* the control graph. By doing so, both the data and control related aspects of a system are completely specified at a high level.

- The use of ADTs in HOP addresses *systems engineering* issues such as reported in [3]. 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.

- Similar to the act of introducing ADTs, HOP allows writing 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*[5].

- HOP's process model is based on the three fundamental operations of hierarchical system design—*composition*, *hiding*, and *renaming*—as identified by Milner[29]. Since HOP is a high-level specification language for synchronous systems, the study of these (and related) operations provides a design theory for synchronous VLSI systems.

- Despite basing HOP on the above elementary mathematical operators, we do not propose that users program directly using these operators. Instead, in the HOP language we provide high level constructs that could be easily translated to a (much larger and relatively very low level) description using these elementary operators. Thus, ease of use as well as formal semantics are both provided.

1.2 Related Work

We compare HOP with other works on two aspects: (i) in its capability to specify complex timing and functional behaviors; (ii) in its capacity to perform PARCOMP, simulation based on PARCOMP, and the detection of control timing errors. Many features of HOP have been omitted here, but have been reported elsewhere[12].

HOP is close in some respects to the work of Milne [26]. The main differences with it are the following:

1. In HOP, value communication has been decoupled from synchronization. The advantages of doing so are discussed in section 2.

2. We emphasis the modeling of value communications and data path state changes in a simple, yet powerful, abstract data type oriented functional language. This is not addressed in [26].

3. HOP adopts a specific timing model—that of lock-step synchronous processes. HOP processes are deterministic. These decisions contribute directly to the simplicity of the language and makes specification driven design more practical.

   On the other hand, Circal includes primitives that are more elementary, and hence more powerful at the expense of being of lower-level.

4. HOP is well suited for describing synchronous hardware systems. A large majority of VLSI systems are synchronous. In this realm, we have conducted a more thorough investigation of many practical aspects of VLSI design.

```
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>
CLOCK <a clock agent and the ports imported from it>
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

HOP is different from more traditional languages (*e.g.* VHDL[18], Karl[30], and ISPS[2]) in many ways, the most important being the following: (i) HOP is much simpler than these languages, and has an equally simple formal semantic definition; (ii) The view of hardware as communicating processes is attractive in many ways than modeling hardware behavior through traditional imperative constructs (procedural or non-procedural descriptions). By creating HOP, we are not discarding or ignoring ongoing efforts towards developing standard HDLs such as VHDL. Our objective is to experiment with interesting ideas not present in such standard languages, and the results may one day benefit future versions of VHDL.

PARCOMP, as well as its planned uses, are similar to the work reported in [15], and to the idea of *constructive simulation* reported in [27]. 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. This is accomplished thus (explained in detail later): "states reachable via transitions labeled by unsynchronized and hidden events are never visited, and consequently the search-space is pruned." Further, we have developed a version of PARCOMP called PARCOMP-DC that can exploit the regularity of vecprocs using a *divide-and-conquer* technique (section 6). 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

The basic unit of specification in HOP is the *module*. The external attributes of a module are:

- Zero or more uni- or bi-directional *data ports*;
- Zero or more uni-directional *events*;
- An external protocol specification.

A module specified as a black box is called an *absproc*, standing for *abstract process*. The skeleton of an ABSPROC is shown in figure 1. A module specified as a network of subprocesses is called a *realproc*, the skeleton of which appears in figure 2. (Note: For ease of

```
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

parsing, currently we use a lisp-like syntax for HOP; we have hand edited *almost* all syntactic descriptions in this paper to an easier-to-understand higher-level syntax.)

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* (figure 3). 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 divide-and-conquer version of PARCOMP has been developed for Vecprocs (section 6).

A realproc is built using one or more absprocs by connecting some of the ports and events of the absprocs, by composing the external protocols of the absprocs, and by internalizing (hiding) some of the events and ports of the absprocs. A syntactically sugered notation (`DATANODE` and `EVENTNODE`) mitigates the burden of specifying the *renaming* and *hiding* ([29]) information for large systems. 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.

We now examine the specification of an absproc in detail.

2.1 Specifying an Absproc

An absproc is specified by its ports, its events, and its protocol.

2.1.1 Ports and Value Communication

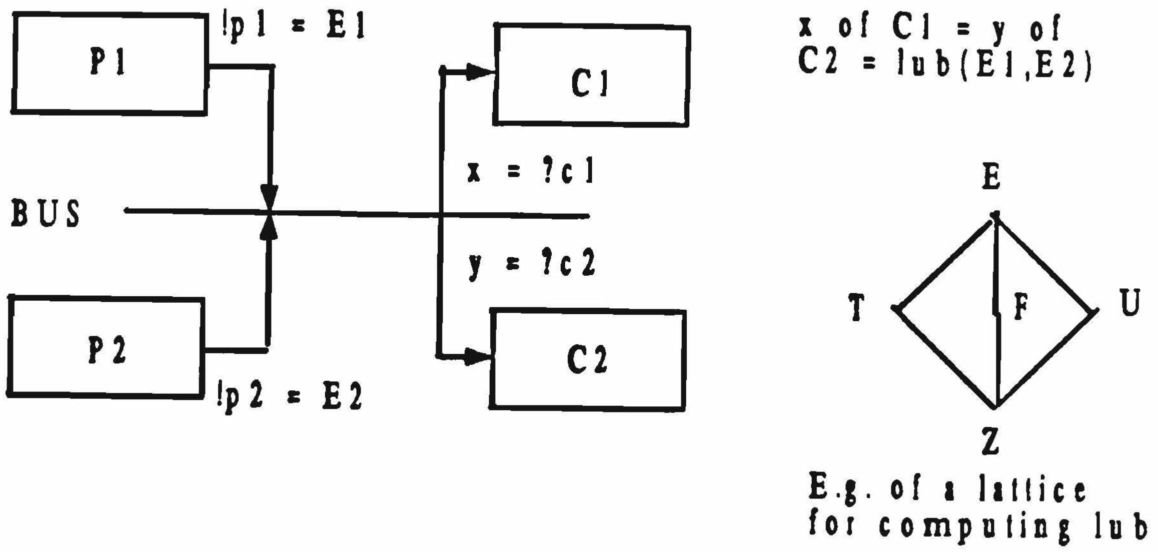


Figure 4: Use of Data Assertions and Queries for Value Communication

The mechanism of synchronized communication as used in [26] does not accurately model the value communication in hardware systems. 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 this system, it is perfectly acceptable to seek the value on the bus while there are no simultaneous writers, or vice versa. (The former case could arise in VLSI where the bus has a "pull-up transistor", for example.) It is even permissible to have two simultaneously active data assertions (say, with compatible "strengths" [5]) on the bus.

In HOP, value communication is performed through a mechanism called *data assertions and queries*. A data assertion, written as `!p=E`, binds an individual variable p representing the *output port* to the value E at the time the data assertion is made. In general, data assertions are of the form `!p=E until e`, where `e` is a future event, where the `until` operator has the same meaning as the until operator of temporal logic. (Events are discussed shortly.) The lack of an assertion can be modeled by the assertion `!p=Z`, where Z denotes high impedance. (For a bus with a pull-up transistor, the assertion `!p=weak1` may be used.)

A data query, written as `x=?q`, binds `x` to the value bound to the *input port* q at the time the query is made. Multiple data assertions (as in bus connections) end up asserting the *least upper bound* (LUB) of the asserted values on the port. For handling multiple data assertions, the type of values communicable via ports in HOP must be organized into a strength lattice [5]. 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`.

The above mechanism of data assertions can be extended for modeling bidirectional devices such as pass transistors, ignoring threshold drops. This is done exactly as done in HOL[6], by asserting that the source and drain nodes are have the same value *if* the gate is held at T. Thus, data assertions and queries permit the *relational style* of specification (i.e. non-directional interactions) for modeling bidirectional devices.

Advantages of Data Assertions and Queries

By having two processes interaction mechanisms (*events* and *data assertions*) we have essentially *separated synchronization from communication*. We now show through an example that this separation is advantageous for hardware modeling. Consider a counter with two commands *reset* and *up* that are triggered via events with the same names. After the counter has been subject to the *reset* event and until it is subject to the *up* event, it asserts a data of 0 on its output port. The process that is responsible for the *reset* and the *up* events can, after it has applied the *reset* event (but before it has applied the *up* event) safely assume that the output will be well-defined (and equal to zero) and sample this output as many times as it wishes, without *any* participation of the counter. In contrast, if value communication were bundled up with rendezvous—as is the case with CSP, CCS, and Circal, the counter would have to actually rendezvous, causing the counter process to make progress in its computation. The writer of the counter process thus has to anticipate all possible places where such rendezvous are possible, and make provisions for them in the specification. Our experience is that this renders hardware specifications unnatural and more complicated. In contrast, with data assertions, once the counter has asserted 0 on its output, it has "discharged all its duties".

The *spirit* in which this extension to communication mechanisms was made, is similar to the extension made by Martin to CSP to include *Probes* [24]. Both these mechanisms show that concurrency constructs developed for concurrent software modeling may not be the best possible ones for hardware modeling.

2.1.2 Events

Events are of two kinds: input, and output. An input event `e` (written `Ie`) denotes a *condition* that a module senses via wires. An output event `e` (written `Oe`) denotes a *condition* that a module generates via wires. Most modules have, at every point in time, a set of events GE ("good events") that would steer the module into well defined modes of activity. Modules also have, at every point in time, a set of events BE ("bad events") for which they do not have any useful behavior defined. We call the GEs at every point in time as the "synchronization points" of the module.

Events help in making implicit synchronization points explicit. For illustration, consider a clocked synchronous system supporting multiple operations. In traditional designs of synchronous systems, the completion of an operation is not explicitly notified, but is *tacitly* assumed after the elapse of a certain interval of time from the start of the operation. However this approach is worse than hard-wiring literal constants in programs leading to programs that are hard to debug or modify. A better approach would be to encourage the writers of module specifications to "highlight" these synchronization points by introducing *events*. These events may be thought of as being implemented by fictitious control and status wires.

Events have a *conceptual reality* even at very early stages of the design; however they attain *implementational reality* (e.g. "should an event be represented in unary, or in binary?", etc.) only much later. The latter decision is influenced by the nature of the controller, and this is typically decided much later in a design life-cycle.

Some of the advantages of using events are:

1. It becomes possible to statically check for sequencing errors. We show some examples in section 4.

2. It highlights the allowed modes of usage of a module. Hardware specifications must not merely attempt to model hardware as it is; rather they must model hardware as *it is expected to be used*. Hardware systems have astronomically more useless combinations of inputs (as well as *sequences* of combinations of inputs) than useful ones.

3. As digital designs evolve, the events that were originally thought to represent fictitious control wires may be implemented as combinations of control signals and clocks. Combinational logic necessary to decode these combinations and raise the corresponding input event will be tacitly assumed, and not modeled explicitly. This is of advantage on two occasions: (i) when these encodings haven't been decided; (ii) in later stages of a design, when these encodings would be excess baggage to carry around.

4. Event connections between modules is achieved via *renaming*. The actual implementation of renaming is through combinational logic that translates a condition in one module to a condition in another. This could pave the way for the synthesis of "glue logic" that connect modules. This connection between a language operator (*renaming*) and its hardware interpretation (*glue logic*) is natural.

2.1.3 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, (and as illustrated by the Roll Back Chip [12]), the use of ADTs having simple definitions can make *reference specifications* far more *reliable* and *easier to understand*.

2.1.4 The Timing Model

Time is a way to order events. In HOP, processes are lockstep synchronous. Therefore the time of every process advances at the same rate, and thus the event ordering we have can be described via three relations: *simultaneous*, *before*, and *after*. A HOP specification may or may not refer to a central clock depending on whether it models a clocked synchronous system or a unit-delay combinational system. Currently we do not have the ability to model some subsystems at the unit-delay combinational level, and the remaining subsystems at the clocked level. We hope to add this capability later on, by specifying clock periods to be fixed integral multiples of unit-delays (an idea proposed in [19]).

In later versions of HOP, we will provide a "clock library", i.e. an expandable library of various clocking schemes. Each entry in this library would specify a clock generator of a certain kind; for instance there would be a *two-phase* clock generator in this library.

2.1.5 An Example of an Absproc: A Pipelined Memory

```
-- 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)                    -- Note-2
                ELSE (update-vector memType m a d) -- Note-3

 read :: m : memoryType, a: addressType -> d : dataType
                IF (> addr memSize)
                   (print "Illegal memory address")
                   (error-obj int)                        -- Note-2
                ELSE (index-vector memType m a)        -- Note-3

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 : error-obj is supported for memoryType by our ADT library
-- Note-3 : index-vector and update-vector supported by memoryType
--          which is defined in ADT Library.
```

Figure 5: Specifications of a Memory

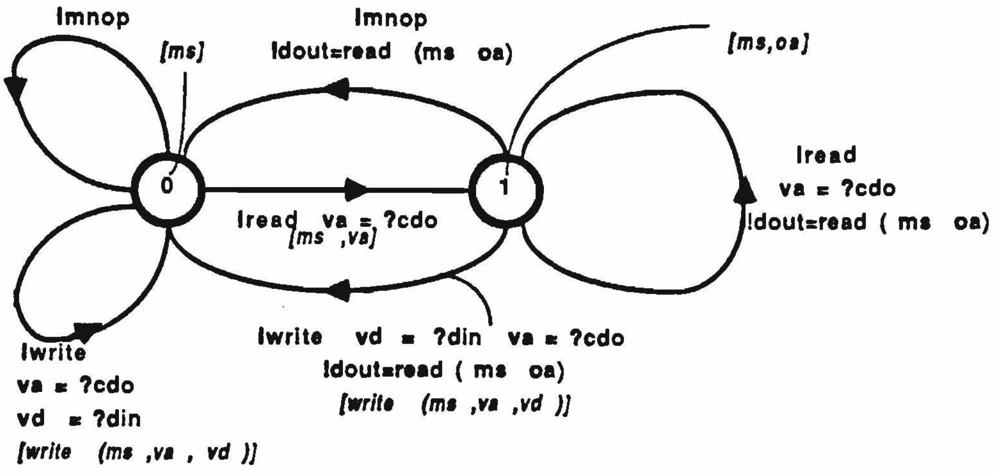


Figure 6: Depiction of the PROTOCOL Specification of MEM

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 "quiescent state", entertain events Inop, Iwrite, and Iread, each of which implement the commands nop (no op), 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 nop (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 assumes that Iwrite, Iread, and Inop are three control wires coming in.

Consider the PROTOCOL section. This section can always 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 notation improves the readability and conciseness of specifications considerably.

The functional expressions *used* in the PROTOCOL section are *defined* in the DEFUN section and/or in the ADT library. Since the ADT library is implemented using object oriented techniques (our technique: "generic types are classes"), functions are overloaded and dispatched correctly. Besides, subtyping is available for free through class inheritance. The data types support both immutable and mutable constructors. We are currently implementing the *in situ evaluation* technique [13] 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 (realized by the unasserted combination of the read and write controls) causes MEM to go back to its top control state; event Iwrite when asserted from outside must be accompanied by data assertions va on the ?addr bus, and vd on the data bus ?din. It causes MEM to go 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 is the pipelining effect). This assertion corresponds to the result of the *previously* requested read. A Iwrite or Imnop takes MEM1 back to MEM; however while reads keep coming, MEM1 goes back to MEM1.

If this memory were to be used in a clocked system, the events Iwrite, Iread, etc. would be generated at the appropriate clock phases. Thus, 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.

We assume that Imnop is a special event that is asserted if none of the other events are asserted. Such an event exists in most modules, and should be defined to be the "unasserted combination of control+clock inputs".

2.2 Specifying Realprocs and Vecprocs

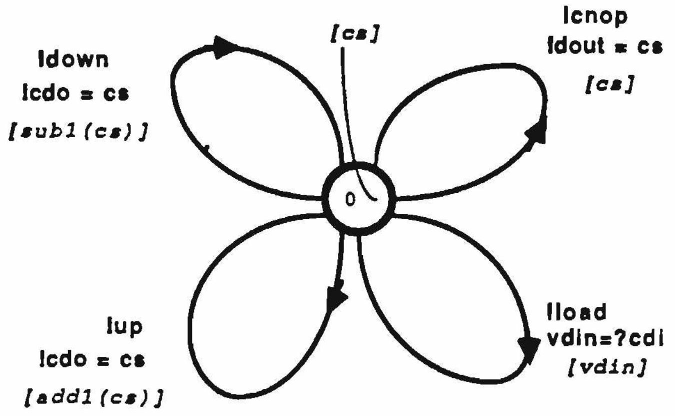
A realproc specifies a system's realization. As an example let us use the memory unit in figure 5 to build a stack using an absproc CTR to implement the stack pointer and a controller SCTL to control the stack. The design of the stack would be specified by writing a realproc specification, as shown in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

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 of a Realproc, 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 ports as well as events among the submodules, and between the submodules and the external ports/events of the stack are specified. Semantically, connections are treated as *renamings*, in the style of [29]. That is, connected entities are renamed to common names that are unique.

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

```
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

-- Note: All the ''nop'' events have to be specified in the present version
-- of HOP. These could be implicit defaults, in later versions.
```

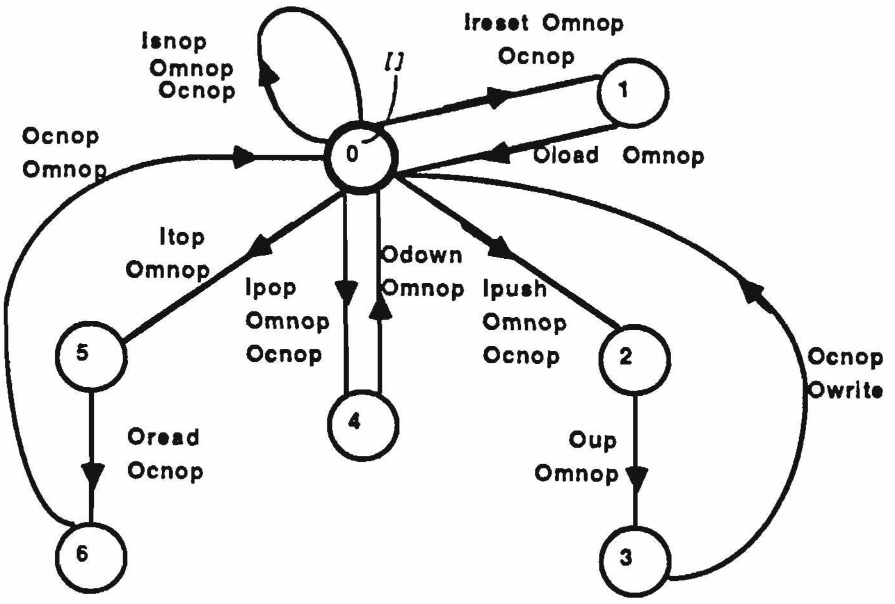


Figure 7: Stack's Submodules:- CTR: An up/down counter; SCTL: Stack Controller

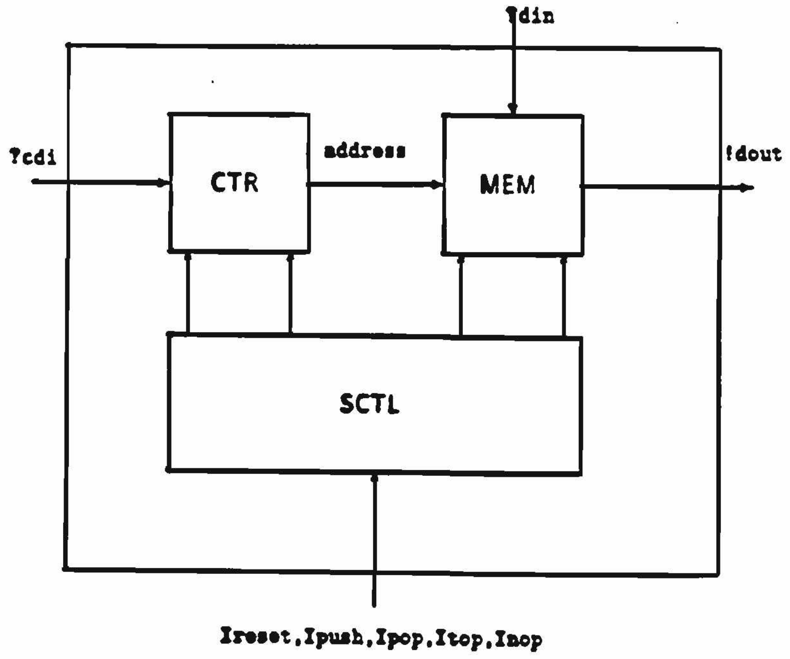


Figure 8: Schematic of the Realproc of a Stack

3 Semantics of HOP

3.1 An Operational Semantics for HOP

In this section, we provide an operational semantics for HOP, using many of the conventions presented by Plotkin [33] for writing operational definitions. In addition to describing HOP unambiguously, these rules form the basis for implementing design tools based on HOP. For instance, PARCOMP is written by following these operational rules. Towards the end of this section, we also briefly touch upon the subject of viewing HOP specifications as Temporal Logic formulae.

It is a common convention when providing semantic definitions to take an *abstract syntax* of the language in question. The (hopefully obvious) translation from the *real syntax* to the *abstract syntax* is not discussed. Also, we do not have space here to summarize the style of writing operational definitions as presented by Plotkin [33], but let us capture the main idea. When writing operational definitions in this style, we try to provide definitions directly using the syntax of the language, through certain "symbol pushing" rules. These rules are to be justified independently using a denotational or axiomatic semantics; however once so justified, the operational "symbol pushing" rules which are usually much simpler can be used in the day-to-day use of the semantics. This is precisely our approach. This is why we have provided a temporal logic based semantics for HOP to match the operational rules presented here. A brief discussion appears at the end of this section. (Readers who find this section hard may cursorily read it.)

The operational meaning of a HOP process is its *transition relation* $\overset{ca}{\rightarrow} = Proc \times act \times Proc$ where the domain of actions for a process is *act* and that of processes is *Proc*. This relation is defined via structural induction using the notation $\frac{ante}{conse}$ where *ante* is an already defined HOP process (the "antecedent"), and *conse* (the "consequent") introduces the next syntactic

```
REALPROC stack [<various size & type parameters>]
PORT
  ?cdi, ?din, !dout : <suitable types>
EVENT
  Ireset, Ipush, Ipop, Itop, Inop = TBD
SUBPROCESS -- Note-4
  MEM : mem [<actual size parameters>]
  CTR : ctr [<actual size parameters>]
  SCTL : sctl

CONNECT
  DATANODE
  -- Note-1
   HIDDEN CONNECTS ((MEM ?cdo) (CTR !cdo))
   ?din CONNECTS ((MEM ?din))
   ?cdi CONNECTS ((CTR ?cdi))
   !dout CONNECTS ((MEM !dout))

  EVENTNODE
  -- Notes-2,3
   HIDDEN CONNECTS ((MEM Imnop) (SCTL Omnop))
   HIDDEN CONNECTS ((MEM Iread) (SCTL Oread))
   HIDDEN CONNECTS ((MEM Iwrite) (SCTL Owrite))
   HIDDEN CONNECTS ((CTR Icnop) (SCTL Ocnop))
   HIDDEN CONNECTS ((CTR Iload) (SCTL Oload))
   HIDDEN CONNECTS ((CTR Iup) (SCTL Oup))
   HIDDEN CONNECTS ((CTR Idown) (SCTL Odown))

   Ipush CONNECTS ((SCTL  Ipush))
   Ireset CONNECTS ((SCTL  Ireset))
   Ipop CONNECTS ((SCTL  Ipop))
   Itop CONNECTS ((SCTL  Itop))
   Inop CONNECTS ((SCTL  Isnop))

END stack
--Note-1: Each line of form <extport>/<hidden> CONNECTS <ports>
--Note-2: Each line of form <extevent>/<hidden> CONNECTS <events>
--Note-3: Currently we have to specify even ''obvious defaults''.
-- Later such defaults (such as unasserted values of events etc.)
-- will be automatically provided.
--Note-4: In general module instance names and module type names
-- are different. Here they are the same. E.g. SCTL and sctl.
```

Figure 9: Realproc of a Stack

$$
\begin{aligned}
Ie,\ Ie &\Rightarrow Ie && (1)\\
Ie,\ Oe &\Rightarrow Oe && (2)\\
Oe,\ Oe &\Rightarrow Oe && (3)\\
Oidle,\ e &\Rightarrow e && (4)\\
!p = E_1,\ !p = E_2 &\Rightarrow\ !p = bus(E_1, E_2) && (5)
\end{aligned}
$$

Figure 10: Definition of *Action Product* in HOP

category of processes that has not been defined so far.

3.1.1 Action Product

Action product captures how simultaneous actions (events and data actions) interact.

An input event Ie represents a logical condition that is awaited (at some time) by a module. An output event Oe represents the assertion of a logical condition at a particular time instant. Event product, written $e1, e2$ captures how two simultaneous events interact.

As an example, the rule $Ie,\ Oe \Rightarrow Oe$ of figure 10 says that if a module awaits an input condition Ie and simultaneously another module *asserts* an output condition Oe, the result is as if Oe is alone produced at that moment. One may ask "what happened to Ie"? The answer is: "it got satisfied by the assertion Oe"; in other words, Ie got synchronized with Oe. This fact, when taken along with the way in which the rules of *Hiding* are defined later, will show us that the process that was awaiting Ie will make progress.

Data actions have only one simplification rule defined for them by action product: when two different data assertions $!p = E_1$ and $!p = E_2$ are made, the resultant value on the port $!p$ is defined by the function $lub(E_1, E_2)$. The lub function computes the least upper bound of its two arguments over a value lattice. (See figure 4 for an example.) A complete definition of the action product operator is given in figure 10.

3.1.2 Definition of the Transition Relation $\xrightarrow{ca}$

In this section, we define the transition relation by structural induction. Before these definitions are applied to a realproc or a vecproc, all the port and event names in their submodules are assumed to be renamed so as to be distinct. Also every compound action used in a definition is assumed to have been reduced to an irreducible form by repeated applications of the action product operator ','.

Process STOP

STOP is the simplest of HOP processes. It has a null transition relation; *i.e.* it always remains halted.

A *finite process* is defined to be one that will become STOP in a finite number of steps. A finite process does not usually represent any practically useful hardware system. Therefore if PARCOMP results in a finite process starting from non-finite processes, there is room for suspicion that there are sequencing errors in the system. When none of the *input* events in the branches of a CHOICE process P are synchronized, and when these input events are all hidden, process P is turned into a finite process. This can happen (for example) due to the erroneous sequencing of control inputs.

Sequential Processes

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

If P is a process, $ca \rightarrow P$ is a process that first performs the compound action ca and then behaves like P. Sequential Processes are a special case of *deterministic choices* where there is exactly one choice available.

Deterministic Choice

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

A process $P = |_i \; ca_i \rightarrow P_i$, where i ranges over an index set I is one that offers a *deterministic choice* consisting of the compound actions ca_i during its first computational step. If choice c_M is accepted, P continues to behave like P_M.

If I has more than one element, then there must be an input event e_i present in each ca_i. Since the e_is govern the selection of one of the alternatives of the choices, the e_is must have pairwise mutually exclusive definitions for their control encodings.

Adding Actions To Initials

If P is a process, cal, P is a process which adds cal to the initials of P.

Add-to-initials: $$\frac{P \xrightarrow{ca} P'}{cal, P \xrightarrow{cal,ca} P'}$$

Hiding

"Hiding an event e" is a shorthand for saying that both Ie and Oe are hidden from a process. Rule *Hiding-sync* considers the hiding of Oe. Oe is replaced by $Oidle$.

Hiding-sync $$\frac{P \xrightarrow{ca} P'}{\texttt{Hide } e \texttt{ in } P \xrightarrow{ca[Oidle/Oe]} \texttt{Hide } e \texttt{ in } P'}$$

The notation "[new/old]" is used to mean that "new" replaces "old".

Hiding Ie from a process prevents it from synchronizing on this event. This can be captured by *pruning* those branches of the synchronization tree that are labeled by Ie:

Hiding-unsync $$\frac{P \xrightarrow{ca1} P',\; P \xrightarrow{ca2} P'',\; e \in cal}{(\texttt{Hide } e \texttt{ in } P) \xrightarrow{ca2} (\texttt{Hide } e \texttt{ in} P'')}$$

Hiding a data output port removes data assertions made on that port from the current compound-action of the process. This would affect those processes that perform a data query from a connected port at the same time:

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

Hiding a data input port causes those variables that would have been bound by a data query on this port to remain unbound:

$$\textit{Hiding-din}\ \frac{P \xrightarrow{ca,x=?p} P'}{\texttt{Hide}\ p\ \texttt{in}\ P \xrightarrow{ca} \texttt{Hide}\ p\ \texttt{in}\ P' with\ x\ free\ in\ P'}$$

Renaming

Processes are made to interact with each other either via events or via data actions (*da*) on ports by renaming their individual event and port names to common names:

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

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

Parallel Composition

The parallel composition operator || models the process of realizing a system by putting together several sub-processes, and permitting their interaction through events and ports that are connected.

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

After performing parallel composition according to the above rule, we may simplify the result by using the following rule (if applicable). This rule captures the effect of value communication:

$$\textit{Value Communication During Parallel Composition}\ \frac{P \xrightarrow{(x=?p),(!p=E),ca} P'}{P \xrightarrow{(!p=E),ca} P'\ [E/x]}$$

Conditionals

HOP processes are usually defined as process schemas $P[dps]$, where for each value of *dps* we have one specific process. *dps* usually represents the data path state of the process. We have the notion of *conditional processes* in HOP that allows us to specify the behavior of a process based on its *dps* variable. Thus we may define a process P as:

$$P[dps] \Leftarrow if\ p(dps)\ then\ P1[f(dps)]\ else\ P2[g(dps)].$$

After reducing the predicate application $p(dps)$ to $true$ or $false$, one of the following rules would apply:

$$\textit{Conditional}\ \frac{P1 \xrightarrow{ca} P'}{(\texttt{if}\ \textit{true}\ \texttt{then}\ P1\ \texttt{else}\ P2) \xrightarrow{ca} P'}\ ;\ \frac{P2 \xrightarrow{ca} P'}{(\texttt{if}\ \textit{false}\ \texttt{then}\ P1\ \texttt{else}\ P2) \xrightarrow{ca} P'}$$

Recursion

A collection of one or more processes may be defined recursively. Since only tail-recursion is allowed, recursion can be modeled as iteration.

3.2 Section Summary

It is possible to view HOP as stylized formulae in Temporal Logic. For instance the specification in figure 11 can be modeled in temporal logic as shown in figure 12.

```
P [s] <=   Ie1 -> !dout = 55 -> P [f(s)]
         | Ie2, x=?din -> Q [g(s,x)]
```

Figure 11: An Example HOP Specification

$$\begin{gathered} P(s) \equiv \Box((Ie1 \supset \bigcirc((!dout = 55) \wedge \bigcirc P(f(s)))) \\ \wedge(Ie2 \supset (x = ?din \wedge \bigcirc Q(g(s,x)))) \\ \wedge(not(Ie1) \wedge not(Ie2)) \supset \Box ERROR). \end{gathered}$$

Figure 12: Temporal Logic Equivalent of the Example HOP Specification

In the Temporal Logic specification, we treat port names *?din* and *!din* as individual variables. Renaming and hiding are modeled in an obvious way. The effect of simultaneous data assertions and queries on a bus can be handled by first computing the LUB of the asserted values (over the value-lattice of the data items asserted), and then binding this LUB to the variables involved in all the queries on this bus.

One benefit of using pragmatically oriented HDLs that have a clean semantics (like HOP), as opposed to directly using universal functional/relational calculii, is simplicity. HOP processes may be viewed as a collection of communicating automatons. The operational semantics provided in this section define the rules of communication, and they may be understood syntactically. Milner [28] and Plotkin [33] have extolled the virtues of this approach.

Another major benefit of using HDLs is the following. Useful "idioms"—commonly occurring patterns in HDL descriptions—can be identified by trying out a large number of examples. Then we can identify a *subset* of Temporal Logic (or another formalism) that matches these idioms. The advantages of identifying such subsets of (*inherently undecidable*) theories is obvious—we can make a focussed attack on the problem of verification and testing of hardware.

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. It works by symbolically simulating all possible interactions between the subprocesses of a realproc or vecproc. PACOMP implements the operational rules of HOP presented in section 3.

The absproc inferred by PARCOMP captures, via symbolic expressions, the behavior of the realproc or vecproc for all possible starting 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 greatly facilitates the understanding of the *collective behavior* of a collection of synchronous systems.

In addition, PARCOMP throws away all of the unused capabilities of a 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, of these 2 operations used, C may sequence them *only in a small number of ways*—out of the myriads of possible ways they may be sequenced. In other words, C implements only some of the astronomically large number of possible microroutines. Such under-utilization of system capabilities is the rule, rather than the exception, in hardware. PARCOMP "distills out" only the used modes of behavior by capitalizing on the *event hiding* information supplied by the designer. Thus, the behavioral descriptions inferred by PARCOMP contain just the right amount of information, and nothing more.

In addition to distilling away unutilized modes of behavior, the *Hiding-unsync* rule reduces the time complexity of PARCOMP. The worst-case time complexity of PARCOMP is proportional to the number of control state tuples actually generated. By pruning away as many control state tuples as early as possible, these control state tuples as well as their successors are never visited.

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.

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 13. (Only the PROTOCOL section of the inferred process is shown.) This description was obtained automatically, using our implementation of PARCOMP. Inferring the behavior of the stack takes less than ten seconds of elapsed time running on an HP-Bobcat running compiled HP Common Lisp.

The inferred PROTOCOL specification asserts that the STACK system offers a choice of events **`Ireset`**, **`Ipush`**, **`Itop`**, **`Ipop`**, and **`Inop`**.

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. During the second tick, it asserts the data value *read*(*ms*, *cs*) on the `!dout` port. This symbolic expression confirms that

```
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]
   | Inop   ->  STACK [cs,ms]
```

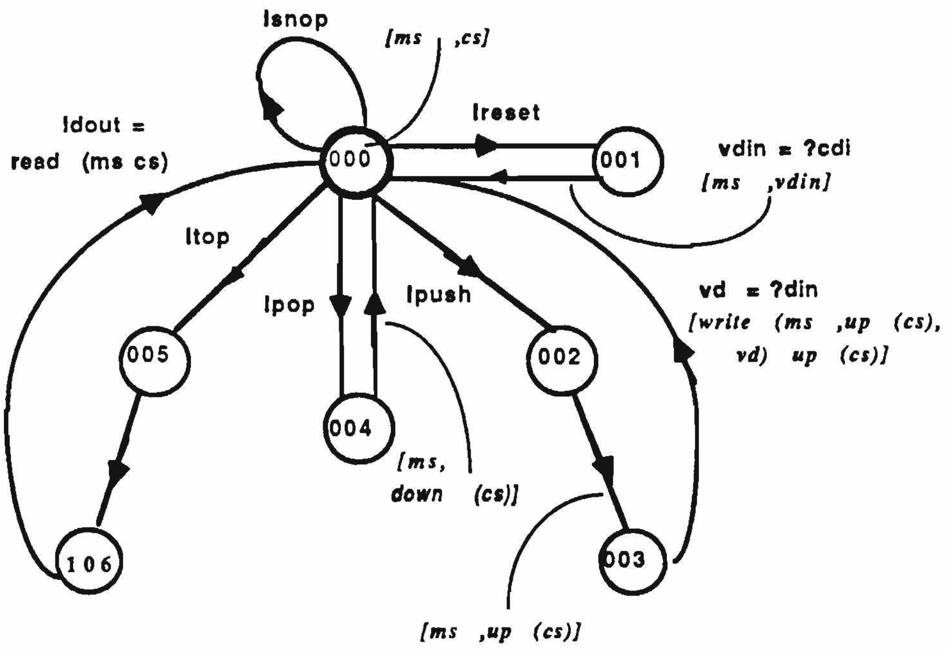


Figure 13: Absproc Automatically Inferred from stkreal using PARCOMP

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 suffer any 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 are similarly correct. (Note: While doing the reset, the initial stack pointer value has to be fed from outside via `?cdi`.)

4.3 How Does PARCOMP Work?

4.3.1 Lockstep Cross-product Automaton

Our explanation of PARCOMP would be greatly facilitated by introducing the concept of *lockstep cross-product automatons.* Given two DFAs A and B, a lockstep cross-product automaton (LCA) of A and B, written $lca(A,B)$, can be obtained from A and B by the following *algorithm*:

(Basis clause): 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 $lca(A,B)$.

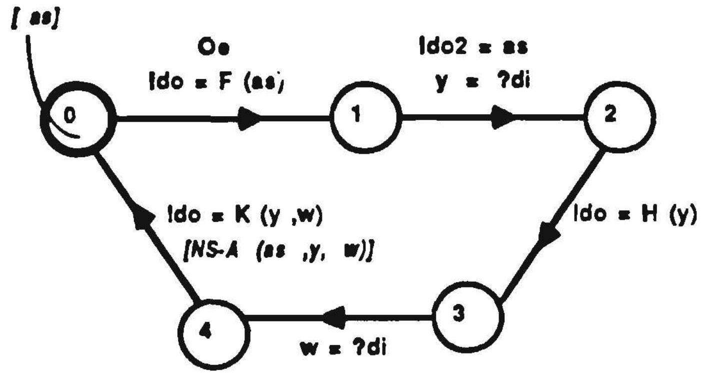
(Inductive clause): If state $< A_i, B_i >$ is in $lca(A,B)$, and there is a directed edge E_{ij} going from A_i to a state A_j in A, (and likewise F_{ij} is a directed edge going from state B_i to a state B_j in B), then $< A_j, B_j >$ is in $lca(A,B)$. Further, the edge EF_{ij} is introduced in $lca(A,B)$ going from $< A_i, B_i >$ to $< A_j, B_j >$.

(Closure clause): There is no other state or edge in $lca(A,B)$.

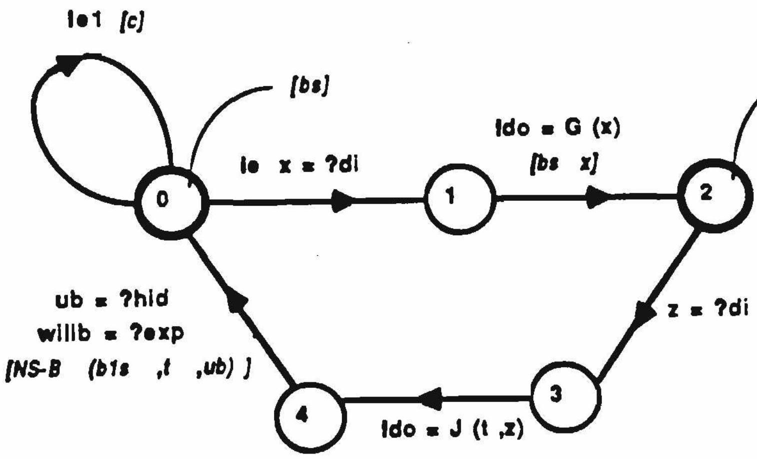
Example: Consider the state diagrams in figure 14 to be DFAs, with state 0 being the starting states of A and B. Then, $lca(A,B)$ contains all the 25 states in the cross-product of A and B. On the other hand if the self-loop at state 0 of process B were to be absent, then it will contain only the five states 00, 11, 22, 33, and 44. The edges in $lca(A,B)$ would then be: $00 \rightarrow 11$, $11 \rightarrow 22$, $22 \rightarrow 33$, $33 \rightarrow 44$, $44 \rightarrow 00$. Thus, we conclude that the number of states in $lca(A,B)$ is less than or equal to the product of the number of individual control states in A and B.

PARCOMP works by attempting to create the LCA. However, as we show below, it actually doesn't create the entire LCA graph—often it creates only a small portion of the LCA graph. In this section, we discuss only the version of PARCOMP that doesn't use the *cond* construct. The *cond* construct is considered in section A.1.

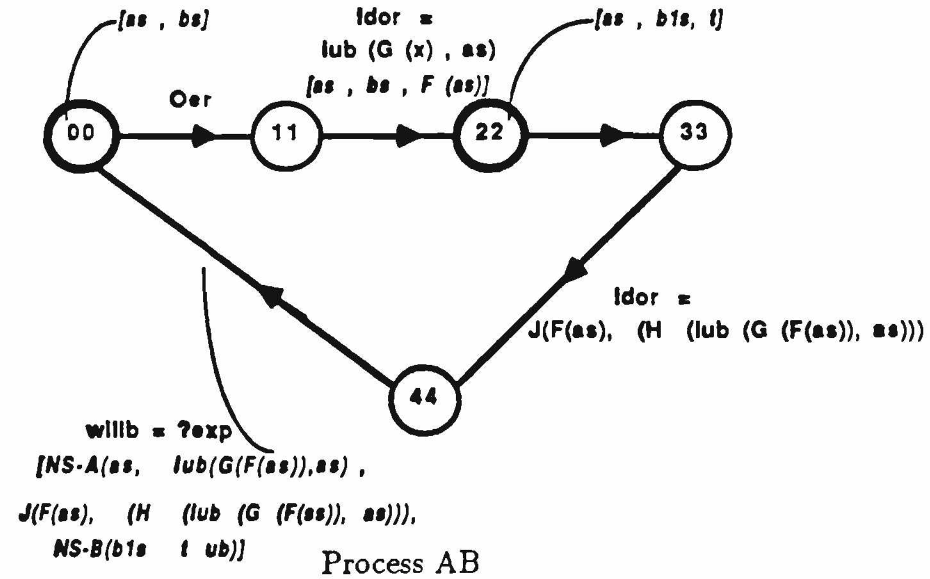
4.3.2 An Illustrative Example



Process A



Process B



Process AB

Figure 14: Processes A, B, and AB

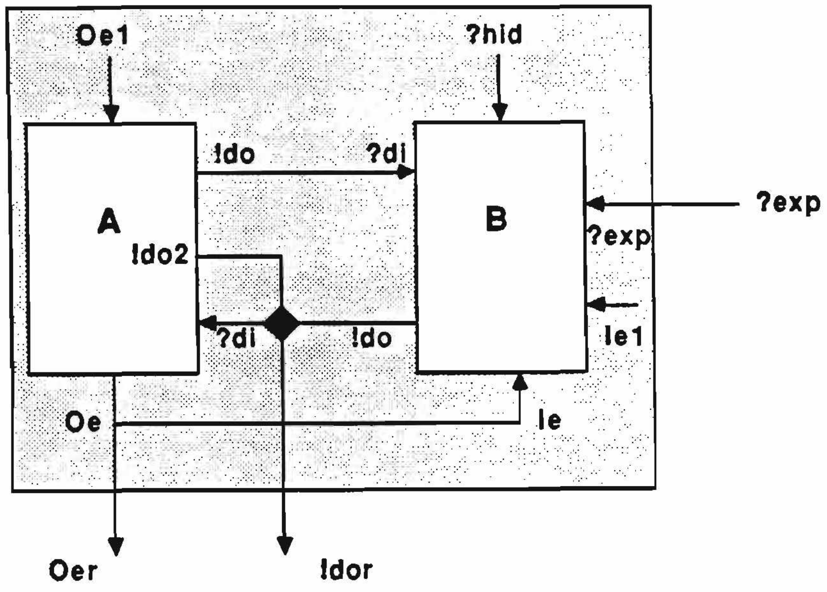


Figure 15: The Realization of the System AB

We illustrate PARCOMP on one example that has been specially constructed to involve most of the interesting cases that arise during PARCOMP. (A rigorous specification of PARCOMP is presented in section A.1.)

The Structural Details

Two processes A and B are connected to form a system called AB, as shown in figure 15. The `Oe1` event of A is unconnected as well as hidden; hence it is effectively ignored throughout. Event `Oe` of A is conneced 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.

The Behavioral Details

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

Figure 14 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 instantaneously jumps 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**. Until it is bound again, the value of variable **y** will represent 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 the explicitly named process "A[as]" in the textual description of A.

Let us consider B. It offers a *deterministic choice* (as explained in section 3) between events Ie1 and Ie in state 0. The former transition will be taken if event Ie1 is asserted (from outside B), *and* event Ie is *not* asserted. The latter transition will be taken if event Ie is asserted, *and* event Ie1 is *not* asserted. (The events guarding the "arms" of a deterministic choice are mutually exclusive, by definition.) 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]. 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 15, 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 14. This behavior was automatically deduced using the PARCOMP procedure.

Operational Rules Invoked in Deducing Process AB

The rules *Renaming-e* and *Renaming-port* of section 3 are used to model connections between ports and events. (In our narration below, we will perform these renamings "as and when needed" during explanation.) Since A and B interact, we invoke the rules *Parcomp* and *Value Communication During Parallel Composition.* Finally we invoke the rules of hiding, to take into account the hidden events and ports.

We now discuss some specific instances of these rules, with respect to figure 14.

- PARCOMP can be thought of as a nested iterative procedure where the outer loop attempts to generate the LCA. The inner loop performs *action products* of events and data queries/assertions, obtaining simplified events and data queries/assertions. These are used to *annotate* the edges of the LCA, thus obtaining the inferred absproc.

  To clarify this, consider the move of B from 0 to 0, and of A from 0 to 1. We obtain the LCA edge $00 \rightarrow 10$. Label this edge with the set of actions obtained from the $0 \rightarrow 1$ edge of A and the $0 \rightarrow 0$ edge of B.

  (Convention: We show these actions prefixed by "A:" if they are caused by A, and "B:" if caused by B. If caused by A and B collectively, we prefix it by "AB:".)

  This compound action is:

  $$B : \mathtt{Ie1},\ A : \mathtt{Oe},\ A : !\mathtt{do} = \mathtt{F(as)}\ \ ---(1)$$

  The other edge in the LCA is $00 \rightarrow 11$, and is labeled by

  $$A : \mathtt{Oe},\ B : \mathtt{Ie},\ A : !\mathtt{do} = \mathtt{F(as)},\ B : \mathtt{x} = ?\mathtt{di}\ \ ---(2)$$

- Consider equation (1). This equation is irreducible under the action product operation (the rules in figure 10). Further, it contains the event `Ie1` that is *unsynchronized and hidden*. This represents a *possible move of B* that will never materialize. So we can invoke the rule *Hiding-unsync*, and prune away this possibility. Thus, we delete the $00 \rightarrow 10$ edge from the $lca(A, B)$.

  This step accounts for the practical efficiency of PARCOMP. In the current example, this one pruning step prevents the generation of the following control state pairs of AB: 20, 30, 40. This is because 20, 30, and 40 are all successors of 10, in the LCA of AB.

- Consider equation (2). It is reducible through equation 2 of figure 10. It reduces to

$$AB : \texttt{0e},\ A : \texttt{!do} = \texttt{F(as)},\ B : \texttt{x =?di} \ --- (3)$$

  This fact represents that `Ie` synchronizes with `Oe`.

  Ports `!do` and `?di` are connected. Since connections are modeled via renaming to a common name, let us rename `?di` to `?do`. Now we can invoke the rule *value communication during parallel composition*, and simplify (3) to:

$$AB : \texttt{0e},\ AB : \texttt{!do} = \texttt{F(as)} \ --- (4)$$

  and also generate the substitution $[F(as)/x]$ to be applied to the "rest of the parallel composition". This shows that the variable x of B would be bound to $F(as)$, thus showing that a value communication has occurred.

- Equation (3) contains `Oe` that is *not* hidden—it connects to the event `Oer` of AB. Thus 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.

- PARCOMP proceeds thus, and *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 "resultant" assertion is computed using the action product rule 3 of figure 10.
  - 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.

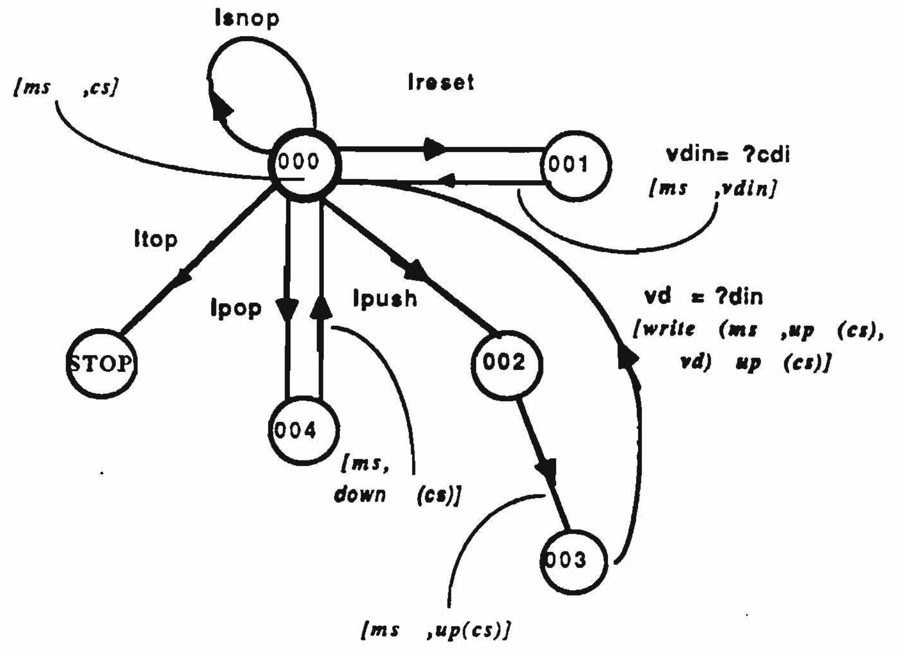


Figure 16: Inferred Behavior of the Stack using an Erroneous SCTL

- 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`. This is obtained formally by invoking the rule *Hiding-din*. 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 could be UB. (UB will be bound to HOP's HIZ value "Z", or to boolean False ("F" in HOP), depending on the actual IC technology used.)

5 Experiments with PARCOMP

In this section we present various experiments that we have 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 16. The inferred

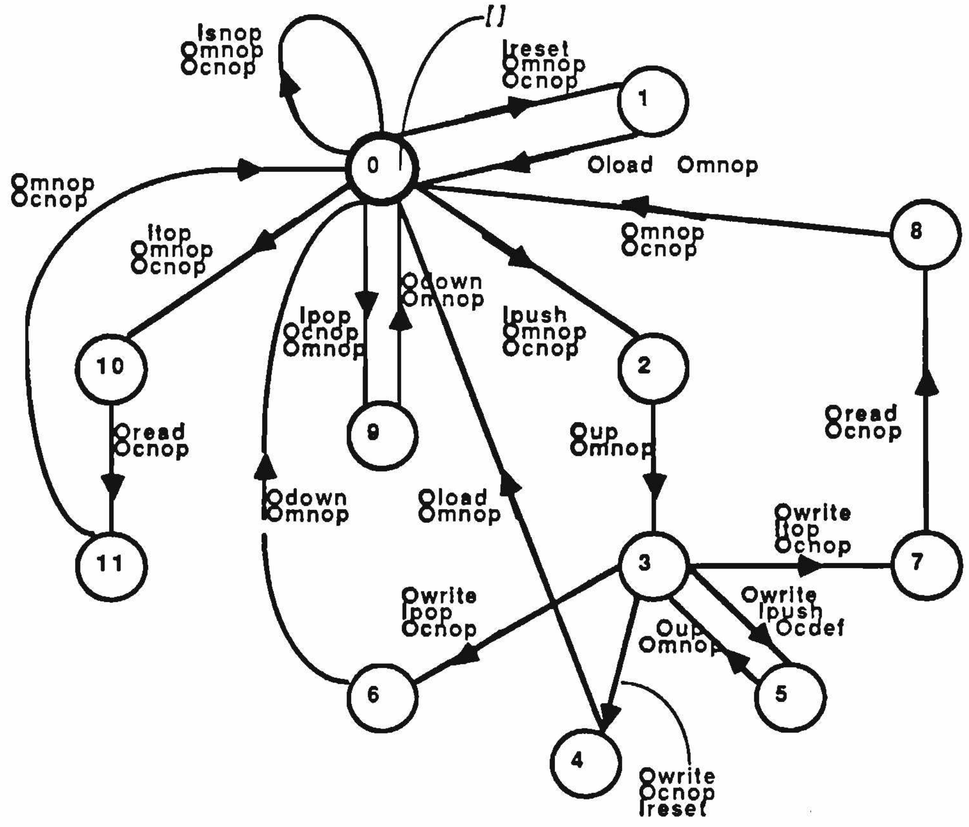


Figure 17: The Pipelined Stack Controller

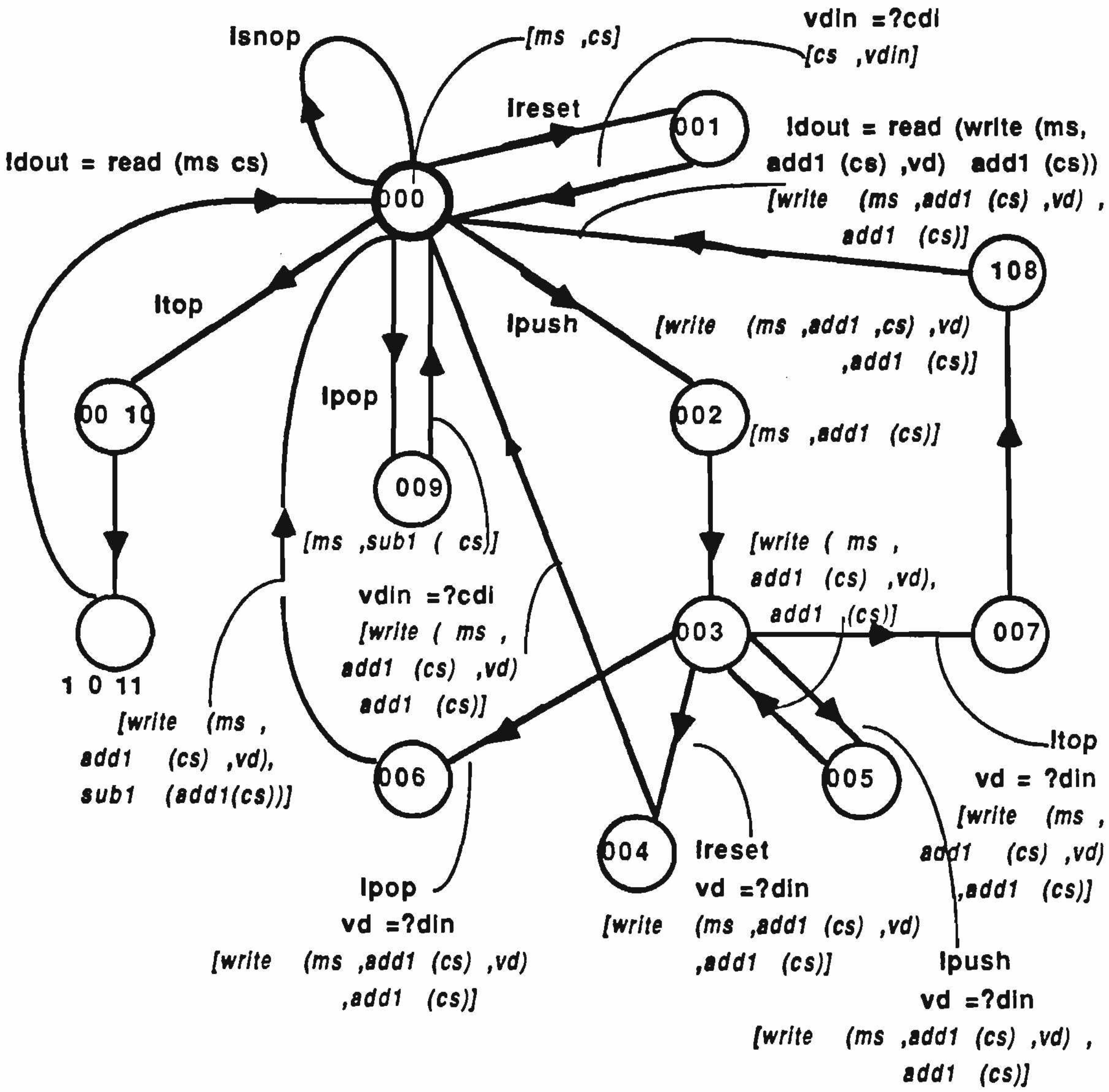
Absproc has a transition from state 000 to state STOP, which is a dead-end. A STOP control state in a process is indicative of a design error, because a hardware system's behavior must be defined for every time instant. Thus when a STOP state is generated during PARCOMP, it issues a warning to the user. 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 will be caught in the above manner. It should be clear that certain errors will not lead to any dead-end control states, but would nevertheless give rise to erroneous modes of behavior.

5.2 Pipelining the Stack

The inferred behavior of the Stack presented in figure 13 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 17. 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`



lsnop
[ms ,cs]
vdin =?cdi
[cs ,vdin]
lreset
001
ldout = read (ms cs)
000
ldout = read (write (ms, add1 (cs) ,vd) add1 (cs))
[write (ms ,add1 (cs) ,vd) , add1 (cs)]
108
ltop
lpush
[write (ms ,add1 ,cs) ,vd) ,add1 (cs)]
00 10
lpop
002
[ms ,add1 (cs)]
009
[ms ,sub1 (cs)]
vdin =?cdi
[write (ms , add1 (cs) ,vd) add1 (cs)]
[write (ms , add1 (cs) ,vd), add1 ,(cs)]
1 0 11
003
007
[write (ms , add1 (cs) ,vd), sub1 (add1(cs))]
006
005
ltop
vd = ?din
[write (ms , add1 (cs) ,vd) ,add1 (cs)]
004
lreset
vd =?din
[write (ms ,add1 (cs) ,vd) ,add1 (cs)]
lpop
vd =?din
[write (ms ,add1 (cs) ,vd) ,add1 (cs)]
lpush
vd =?din
[write (ms ,add1 (cs) ,vd) , add1 (cs)]

Figure 18: Inferred Behavior of the Pipelined Stack (one that uses PCTL)

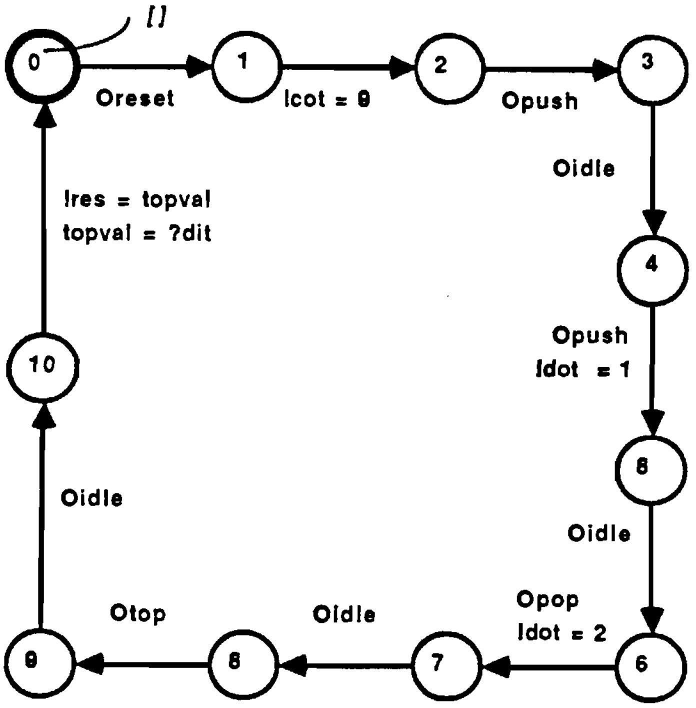


Figure 19: A Tester Process for the Pipelined Stack

operation.

Using PCTL and the same old MEM and CTR, PARCOMP infers the behavior shown in figure 18. 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 19.

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 20. This approach has many practical advantages, and they are discussed in the following subsections.

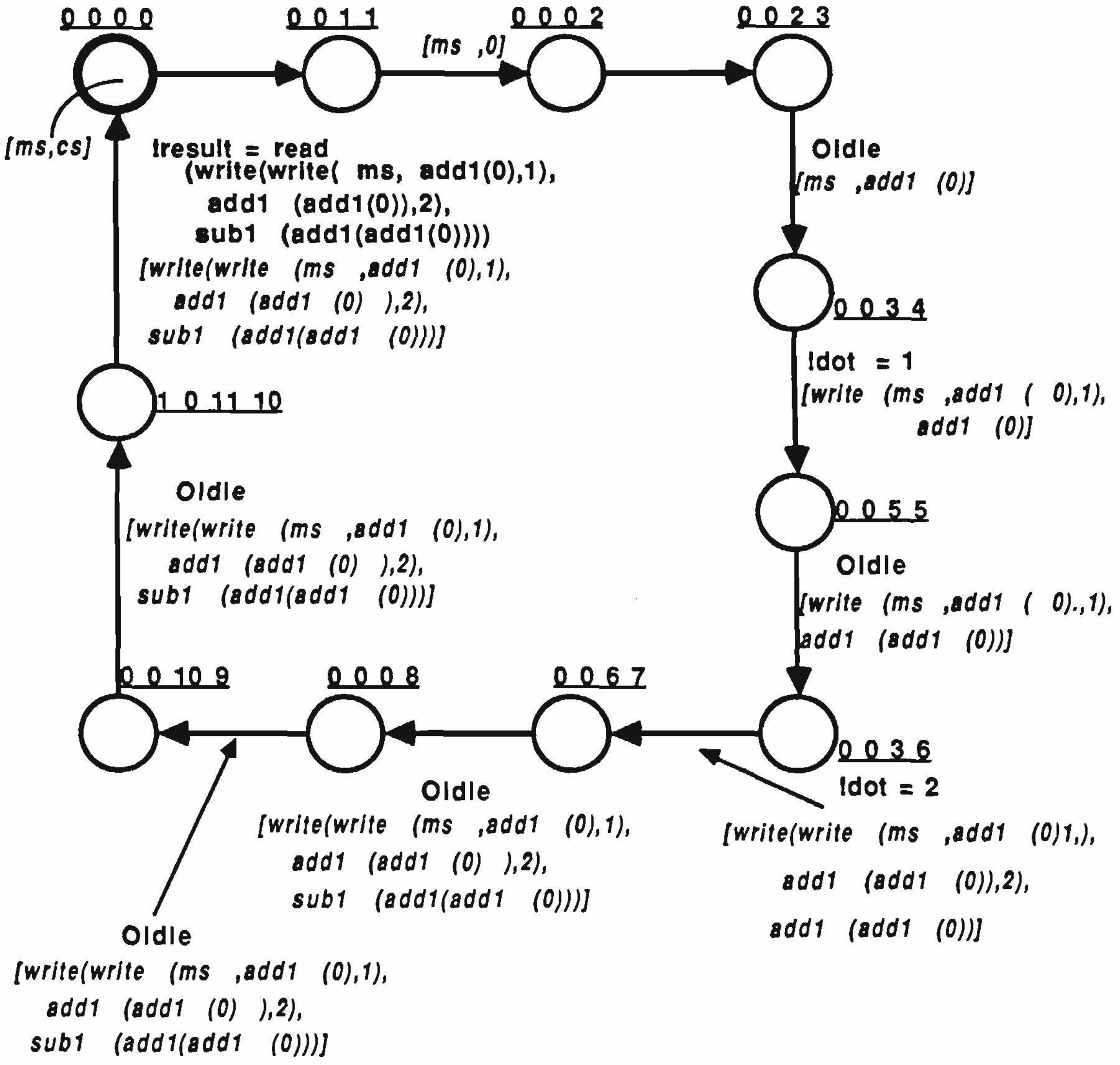

0 0 0 0
0 0 1 1
[ms ,0]
0 0 0 2
0 0 2 3
[ms,cs]
!result = read (write(write(ms, add1(0),1), add1 (add1(0)),2), sub1 (add1(add1(0))))
[write(write (ms ,add1 (0),1), add1 (add1 (0)),2), sub1 (add1(add1 (0)))]
OIdle
[ms ,add1 (0)]
0 0 3 4
!dot = 1
[write (ms ,add1 (0),1), add1 (0)]
1 0 11 10
OIdle
[write(write (ms ,add1 (0),1), add1 (add1 (0)),2), sub1 (add1(add1 (0)))]
0 0 5 5
OIdle
[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
OIdle
[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))]
OIdle
[write(write (ms ,add1 (0),1), add1 (add1 (0)),2), sub1 (add1(add1 (0)))]

Figure 20: Composition of the Tester and the Testee (the pipelined Stack)

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 20 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:

```
!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.

We also have developed the prototype of a *compiled simulator* that compiles "tester+testee" processes into procedural code. This simulator is called the CAPS (Compiled AbsProc Simulator). (Note: Some of the above ideas may be found in [15] also.)

5.3.3 Building Partial Testers

Suppose we want to supply certain test stimulii "automatically" from the tester process and some other test stimulii 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 19. Running PARCOMP on this "tester+testee" would result in an "unsatisfied but un-hidden" data query at time 4. When we run CAPS 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 CAPS simulation will run on its own, without user intervention. At the other extreme, the tester would do *nothing*, and the CAPS 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).

In the RIPS simulator, the tester and testee are run concurrently, and the action products are computed at run-time. RIPS is relatively more inefficient than CAPS; however, RIPS allows many flexible interactions not possible with CAPS. For example, after a few simulation steps, we can selectively ignore a subset of the modules, and carry the other modules forwards in simulation. Or, we can add an extra process after a few steps.

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 19, 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 [23] 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
```

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. Some details appear in appendix A.2.

6 A Divide-and-conquer PARCOMP, PARCOMP-DC

This section shows how we can often reduce the run-time of PARCOMP by exploiting the fact that it is commutative and associative.

Consider the array A shown in figure 21. It consists of a collection of modules M connected in a regular interconnection pattern. For simplicity of explanation, assume a nearest-neighbor connection that is regular in both the dimensions. Consider the problem of computing $PARCOMP(A)$; *i.e.* the composition of all the Ms constituting A. Since $PARCOMP$ is both commutative and associative, we can split A into two halves, say A_T standing for "the top of A" and A_B, standing for "the bottom of A", and assert:

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

Since A_T and A_B differ only in the names of their external ports, we need compute *only* $PARCOMP(A_T)$. $PARCOMP(A_B)$ can be obtained from this, by 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 21.

PARCOMP-DC is often more efficient than PARCOMP. Let us make an approximate cost analysis.

As discussed in section 4.3.1, the worst-case time complexity of PARCOMP is proportional to the cross-product of the number of control states in each of the processes, assuming that the

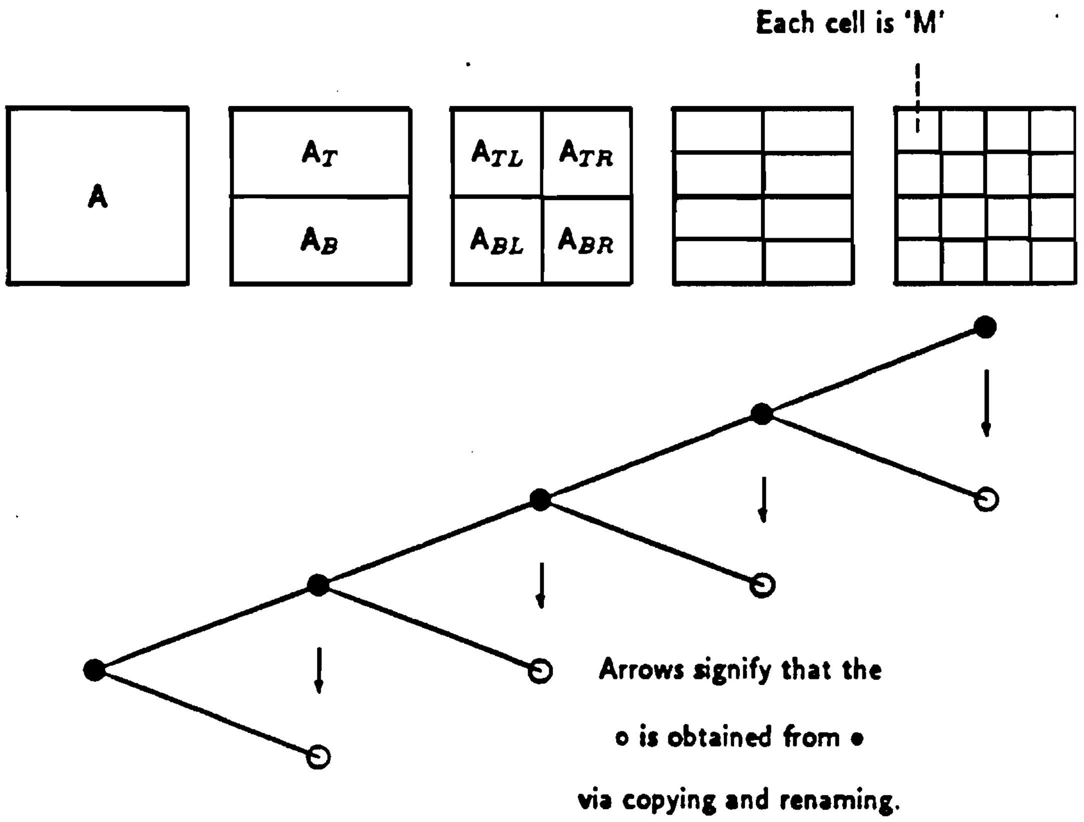


Figure 21: Divide and Conquer PARCOMP

number of events and data assertions on every transition are bounded by a constant. Suppose for simplicity that array A is square, and has N modules of type M, M has C control states in it, and that N be a power of 2. Then

$$cost\_parcomp(A) = O(C^N).$$

Suppose that the modules formed during the division process of PARCOMP-DC are M, ..., A_{TL}, A_T, A. Let $ncs(M)$ denote the number of control states in a module M. Further let *C_copying* denote the cost of copying the process descriptions (see figure 21). Then

$$cost\_parcomp\_dc(A) = O(ncs(M)^2 + \ \dots\ + ncs(A_{TL})^2 + ncs(A_T)^2 + ncs(A)^2 + C\_copying).$$

The above sum has $log_2(N)$ terms. Let D be the ***root mean square*** (RMS) value of the number of control states in M, ..., A_{TL}, A_T, A. Let the cost of copying and applying renamings to a process description not exceed the number of control states in it.

Then,

$$cost\_parcomp\_dc(A) = O(\log_2(N) \times (D^2 + D^2)) = O(\log_2(N) \times D^2).$$

Firstly we note that D does not tend to increase as the size of the modules grow. This is a fact of practical systems because when designing a module using several submodules, only very few of the astronomically large number of sequences of the submodule operations are actually used. Hence the number of control states in a module is often vastly smaller than what it could be. (Consider for example the total number of possible microprograms for a typical datapath .vs. the number of microroutines that are actually ever used!) Thus if D is close to C and if M is large, then there is a significant payoff by using PARCOMP-DC.

In conclusion, the following additional avenues of research are available for handling geometrically regular, (but perhaps computationally irregular—or arhythmic) arrays:

- Perform PARCOMP of two modules of the array;
- Study the inferred behavior and see if it is verifiable manually or through exhaustive simulation.
- The behavior inferred by PARCOMP (or PARCOMP-DC) will have complex if-then-else functions. Construct tabular functions corresponding to these.
- Use these tabular functions for efficient simulation.
- Try to perform formal verification of the whole array by setting up an induction.

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.2 presents an overview of this system. It has a working prototype, currently written in Common Lisp and FROBS [31]. Though we have taken simple examples in this paper, we have worked out some larger examples as well. Some of these hardware units are discussed in [12]; many are yet to be published. Links to VLSI design are briefly described in section A.2.

References

[1] 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.

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

[3] Frederick P. Brooks. *The Mythical Man-month.* Addison-Wesley, 1975.

[4] 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.

[5] Randall E. Bryant. A Switch Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computer*, C-33:160–177, February 1984.

[6] 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.

[7] 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.

[8] Avra Cohn. Correctness Properties of the Viper Block Model: The Second Level. In *1988 Banff Workshop on Hardware Verification*, Springer Verlag, 1988.

[9] 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.

[10] 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).

[11] Ganesh C. Gopalakrishnan. Synthesizing Synchronous Digital VLSI Controllers Using Petri nets. In *International Workshop on Petri Nets and Performance Models, Madison, Wisconsin*, August 1987.

[12] Ganesh C. Gopalakrishnan, Richard M. Fujimoto, Venkatesh Akella, N.S. Mani, and Kevin N. Smith. Specification Driven Design of Custom Architectures in HOP. 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.

[13] Ganesh C. Gopalakrishnan and Mandayam K. Srivas. Implementing Functional Programs Using Mutable Abstract Data Types. *Information Processing Letters*, 26(6):277–286, January 1988.

[14] 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.).

[15] 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.

[16] Matthew Hennessy. *Proving Systolic Systems Correct.* Technical Report CSR-162-84, Department of Computer Science, University of Edinburg, June 1984.

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

[18] April 1986. Special Issue on the VHDL Language. IEEE, Design and Test.

[19] I.S.Dhingra. Formal Verification of a Design Style. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 293–321, Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

[20] Steve Jacobs and Kent Smith. TILER User's Guide. 1986. User's Manual Available from the Univ. of Utah, Dept. of Computer Science VLSI Group.

[21] 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.

[22] 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.

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

[24] Alain J. Martin. The Probe: An Addition to Communication Primitives. *Information Processing Letters*, 20(3):125–130, April 1985. An Erratum related to this article appeared in the August 1985 issue of the Info. Proc. Letters.

[25] 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.

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

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] Gordon D. Plotkin. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, Aarhus University, Denmark, September 1981.

[34] R.C.Sekar and Mandayam Srivas. Specification and Verification of the Lilith Microprocessor in SBL. In *Banff Hardware Verification Workshop, June 1988*, 1988.

[35] 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.

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

[37] 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.

[38] W.F.Clocksin. Logic Programming and Digital Circuit Analysis. *Journal of Logic Programming*, (4):59–82, 1987.

A Appendix

A.1 A Specification of PARCOMP

$\boxed{\textit{Input:}}$ An expression `Hide` $HS\ in\ \ \| \ \{P_i[\overline{X_i}], ..., \ C_j[\overline{X_j}], ...\}$ for $i \in \{1..m\}$, $j \in \{1..n\}$. C_j are conditional processes of the form
$C_j[\overline{X_j}] =$ `if` q_j `then` $T_j[g_j(\overline{X_j})]$`else` $F_j[h_j(\overline{X_j})]$ and P_i are non-conditional processes of the form
$P_i[\overline{X_i}] = y_i\ :\ \textit{initials}_i \rightarrow R_i(y_i);$

Each P_i offers a set of initial choices $\textit{initials}_i$ and for each choice y_i that is offered, the future behavior of P_i is $R_i(y_i)$. HS is the *Hidden Set*, the set of events and ports hidden from the parallel composition.

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

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

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

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

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

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

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

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

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

- Consider $\| \ \{P_i[\overline{X_i}], ...\}$. Let each P_i be

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

-

- Generate tuples

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

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

  - those tuples T that contain ***unsynchronized*** events Ie that belong to HS are dropped, and the corresponding arm of the synchronization tree is pruned;
  - those tuples T that contain Oe that belong to HS are replaced via the substitution $T[Oidle/Oe]$.

- In computing

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

  the bindings generated by taking action products of the members of T are taken into account. □

A.2 A Brief Description of the HOP Design System

Figure 22 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 the PPL[33, 37] layout editor called Tiler [20]. (VLSI chips will be described in PPL; see [12] for links between HOP and PPL.) 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 inferred behavior will be much faster to simulate.

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 [25]. 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.

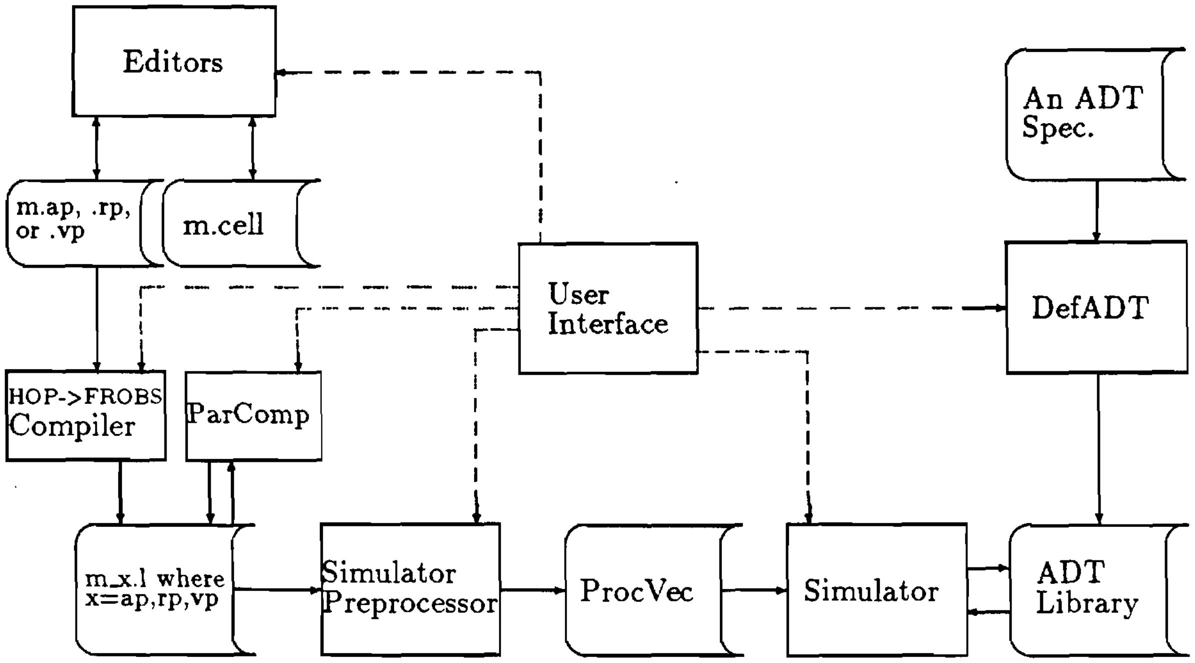


Figure 22: Data Flow Diagram of the HOP Design System