Specification-Driven Design of Custom Hardware in HOP

Ganesh C. Gopalakrishnan, Richard M. Fujimoto,
Venkatesh Akella, N. S. Mani, Kevin N. Smith

UUCS-88-018
October 12, 1988

Specification-driven Design of Custom Hardware in HOP

Ganesh C. Gopalakrishnan, Richard M. Fujimoto,
Venkatesh Akella, N.S.Mani, and Kevin N. Smith
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. Processes may be described both as a black-box and as a collection of interacting sub-processes. The latter can be statically simplified using an algorithm 'PARCOMP'. PARCOMP symbolically simulates a collection of interacting processes. The advantages claimed for HOP include simple semantics, intuitiveness, high expressive power, and numerous provisions to support easily verifiable designs all the way to VLSI layout.*

After introducing HOP, and presenting some of the results obtained from experimenting with the HOP design system, we present the design of a large hardware system (the "Utah Simulation Engine") currently being developed to speed-up distributed discrete event simulation using Time Warp. Issues in the specification driven design of this system are discussed and illustrated using HOP.

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 [JBB88,She85] Prolog [WFC87], Petri Nets [Chu87], Temporal Logic [BCDM85], various Calculii of Communicating Systems [Mil83,Hen84] Trace Theory [Sne85], Higher Order Logic [CGM86,JB85], Algebraic Specifications and Equational Techniques [GSS87,Sub83,NS88], Synchronized Transition Systems [GGS88], and Path Expressions [ACFM85], 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 remain unsolved.

Our research is aimed at solving some of these problems. We have designed a simple hardware description language called HOP (Hardware viewed as Objects and Processes) that embodies solutions to these problems. HOP deals with the structure, behavior, and timing of digital hardware systems. We are also using HOP to specify VLSI circuits that are designed and implemented by various student groups. One such design called the Roll Back Chip (RBC) is described in this paper. Our research thus involves developing HOP as well as using it in many ways for designing real-life hardware.

Our effort to date has given us the following insights, as well as specific results.

Insights

Complex designs evolve in several dimensions. When designing custom computer architectures, many major design decisions are taken over a prolonged duration of time. Some of these decisions are: (i) should a computation be implemented in hardware, firmware, or software? (ii) should a hardware unit be asynchronous or synchronous? (iii) how should the data and control interactions be organized? The problem of effective communication among the various hardware designers of a project (and also between the same designer, over many days!) is often as severe as reported in [Bro75].

There is no satisfactory design language that specifies a finished hardware architecture completely and formally, or accomodates the evolving nature of the design of a custom computer architecture and helps facilitate communication among designers. Although some impressive work has been reported, (e.g. [Bae86]), most works either focus on control flow and ignore data flow, or ignore resource utilization, or ignore complications such as interrupts, exception handling, etc.

What we require is a simple and semantically well understood hardware description language (HDL) that can be used to capture the *structural*, *behavioral*, and *temporal* aspects of an evolving hardware+software design. In addition, a VLSI design system (an assemblage of tools) that supports the simulation and validation of descriptions in such a language is also inevitable [Seq87].

It has been reported that the complete formal verification of even extremely simple ICs is really hard [Coh88]. 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. Until this situation changes dramatically, custom architectures would at best partially proven correct for *certain* critical safety properties. Therefore formal specification will be used to a large extent for its *indirect benefits*—better understanding, better communication among hardware designers and system software writers, better testing, the ability to measure what exactly has been tested, etc.

Given this, it is unquestionable that well designed HDLs, that help the designers *think effectively* using high-level abstractions and thereby intuitively establish the correctness of designs, have a major role to play. To paraphrase Stoy [Sto77, Page 7], "well designed HDLs of the above nature will help write specifications that will more probably be correct because we will less likely have forgotten about the crucial little exception to some general rule that applies in our particular case."

Finally, computations are often done in custom hardware for gaining speed. Thus, lack of throughput is also a design error. Hardware Verification does not yet address performance issues.

Specific Results

1. We have designed a version of HOP that can describe *lockstep synchronous processes*. Specific instances of such processes are: (i) synchronous hardware systems; (ii) hardware systems studied under the unit-delay timing model. (Note: In this paper the language "HOP" refers to the lockstep synchronous version of HOP.)

2. We have described the semantics of HOP both operationally as well as through linear-time temporal logic.

3. We have discovered several new techniques for simulating and debugging (via static analysis) specifications. These procedures have been implemented, and experimental results are presented.

4. An implementation the HOP VLSI design system is in progress, with some parts already operational.

5. We have designed a number of VLSI chips, and have also specified them in HOP.

Organization of the Paper

Section 2 presents the HOP language. Section 3 presents the semantics of HOP. Section 4 presents the HOP design system, and some results obtained to date. Section 5 presents the specification-driven design of the RBC. Section 6 presents the detailed design of one of the submodules of the RBC. Section 7 presents our concluding remarks, and an outline of the planned extensions to HOP. The Appendix describes the algorithms PARCOMP and PARCOMP-DC.

Related Work

The HOP *language* is inspired in many ways by the work of Milner [Mil82] and Milne [Mil83]. The SBL language designed by the first author under Smith and Srivas also [GSS87,Gop86] influenced HOP. Our work is different from related work in these respects: (i) we have focussed on lock-step synchronous timing model,thus obtaining results useful for synchronous hardware; (ii) we model value communications using *data queries* and *data assertions* (as opposed to existing ways of value communication in CSP or CCS); doing so has several advantages, to be discussed; (iii) our work addresses a number of practical design issues in a formal framework.

Broadly speaking, there are two kinds of design automation researchers: (a) Those who take one formalism (such as lazy functional languages or HOL), "piously" believe in it, and "get the most mileage out of it", at the risk of making some practically unrealistic assumptions; (b) those who don't carry such pious beliefs, but treat all formalisms with equal interest, use them as "tools", assimilate group (a)'s results and render them more practical. We belong more in group (b) than in group (a).

```
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

```
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

2 The HOP Language

The basic unit of specification in HOP is an *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 parsing, currently we use a lisp-like syntax for HOP; we have

```
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

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 regular, or rhythmic, as in systolic arrays. Previous work involving regular arrays ([She84,She85], [Pat85], [MH85]) has dealt more with systolic arrays than with arythmic arrays. We will show that having the special category of Vecprocs is useful in many ways.

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* ([Mil80]) information for large systems. A vecproc is essentially built in the same fashion; however a notation based on recurrence relations is provided for easily specifing the regular placement of modules as well as regular interconnections among them.

An algorithm called PARCOMP ("Parallel Composition") has been designed. It 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 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.

PARCOMP, as well as its planned uses, are similar to the work reported in [HK87], and to the idea of *constructive simulation* reported in [Mil85]. 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. The complexity analysis of PARCOMP-DC shows that it could often be faster. Finally, PARCOMP can 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.

Due to the availability of PARCOMP, it is helpful to think of HOP realprocs and vecprocs as having only absprocs as their submodules (*i.e.* if the submodules are themselves realprocs or vecprocs, one could infer equivalent absprocs using PARCOMP).

We now examine the specification of an absproc in detail.

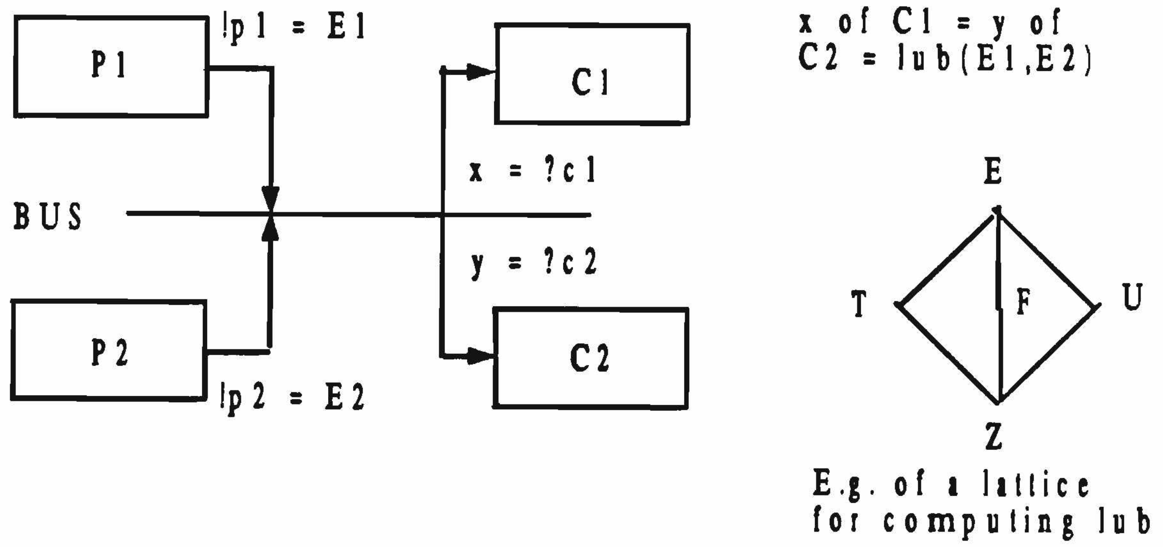

!p1 = E1
P1
C1
x = ?c1
BUS
y = ?c2
P2
C2
!p2 = E2
x of C1 = y of C2 = lub(E1,E2)
E
T
F
U
Z
E.g. of a lattice for computing lub

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

2.1 Specifying an Absproc

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

2.1.1 Ports and Value Communication

The mechanism of synchronized communication as used in [Mil83] 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 have a query without a simultaneous assertion, or vice versa. It is even permissible to have two simultaneously active data assertions (say, with compatible "strengths" [Bry84]) 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.

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 [Bry84]. 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.

The mechanism of data assertions and queries is satisfactory for modeling bidirectionality and bussing at the architectural level where "sufficient time is given for combinational paths to settle". Pass transistors can be modeled as idealized switches, ignoring the threshold drop (as in HOL[CGM86]). Busses can be semantically modeled as *logical variables* [Lin85] (as pointed out to us by the author of the cited paper). By having two processes interaction mechanisms (*events* and *data assertions*) we have essentially *separated synchronization from communication.*

Advantages of Data Assertions and Queries

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 is 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".

One may complain that the separation of communication from synchronization is error-prone. In our experience, this is not true. We have written specifications of systems using data assertions as well as traditional synchronization-plus-communication constructs; the former proved to be much more elegant for modeling value communications hardware. The *spirit* in which this extension to communication mechanisms was made, is similar to the extension made by Martin to CSP to include *Probes* [Mar85]. Both these mechanisms show that concurrency constructs developed for software may not be the best possible ones for hardware modeling.

Data assertions and queries have well-understood and simple semantics. They can model "thru connections" (to be discussed shortly). Interactions such as shown in figure 4 are too awkward to model using existing "CCS/CSP like" languages.

2.1.2 Thru Connections

Now we turn our attention to an important (but hitherto neglected) class of ports called *through connections.* Through connections are wires that pass through a module with or without touching an internal node. It is not satisfactory to model a through connection as a node resident outside the module because by doing so the correspondence between the layout and the high-level specification is not maintained at module boundaries. Maintaining this correspondence would simplify the implementation of a VLSI design system considerably, as has been our experience so far in our use HOP to model Path Programmable Logic (see section 6). In addition, we believe that mixed-mode simulation, layout synthesis, formal verification etc. would be greatly facilitated by maintaining identical interfaces for modules at all levels.

Information regarding through connections is used while compiling a realproc and a vecproc, as well as during PARCOMP. Often thru connections result in bus structures that are embedded within modules. These effects of thru connections are handled appropriately.

2.1.3 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 pleasant.

2.1.4 Data Path States

In the specification of an absproc, the data path state of the system being specified can be modeled using an appropriate high-level ADT. In our experience, (and as illustrated by the Roll Back Chip (section 5), the use of ADTs having simple definitions can make *reference specifications* far more *reliable* and *easier to understand.*

The introduction of new abstract data types into HOP is greatly facilitated by using an underlying object oriented language called FROBS [Mue87]. The class mechanism of FROBS is used to implement *generic types* (such as the class of stacks over various sizes and element types). Each class acts as a repository of the attributes of the type concerned. Subtyping is realized through class inheritance. The creation of an instance of a generic type amounts to creating an instance of a class. The values of a type themselves are implemented as "(type-descriptor . lisp-data-structure)" pairs. (Overloadable) class methods implement the data type operations. These decisions have made HOP's ADT library very well organized.

2.1.5 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 [ISD88]).

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.

2.1.6 An Example of an Absproc: A Pipelined Memory

Consider memory module `MEM` which has an address input port `?addr`, a data input `?din` port, and a data output port `!dout`. It can, in its "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

```
-- 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

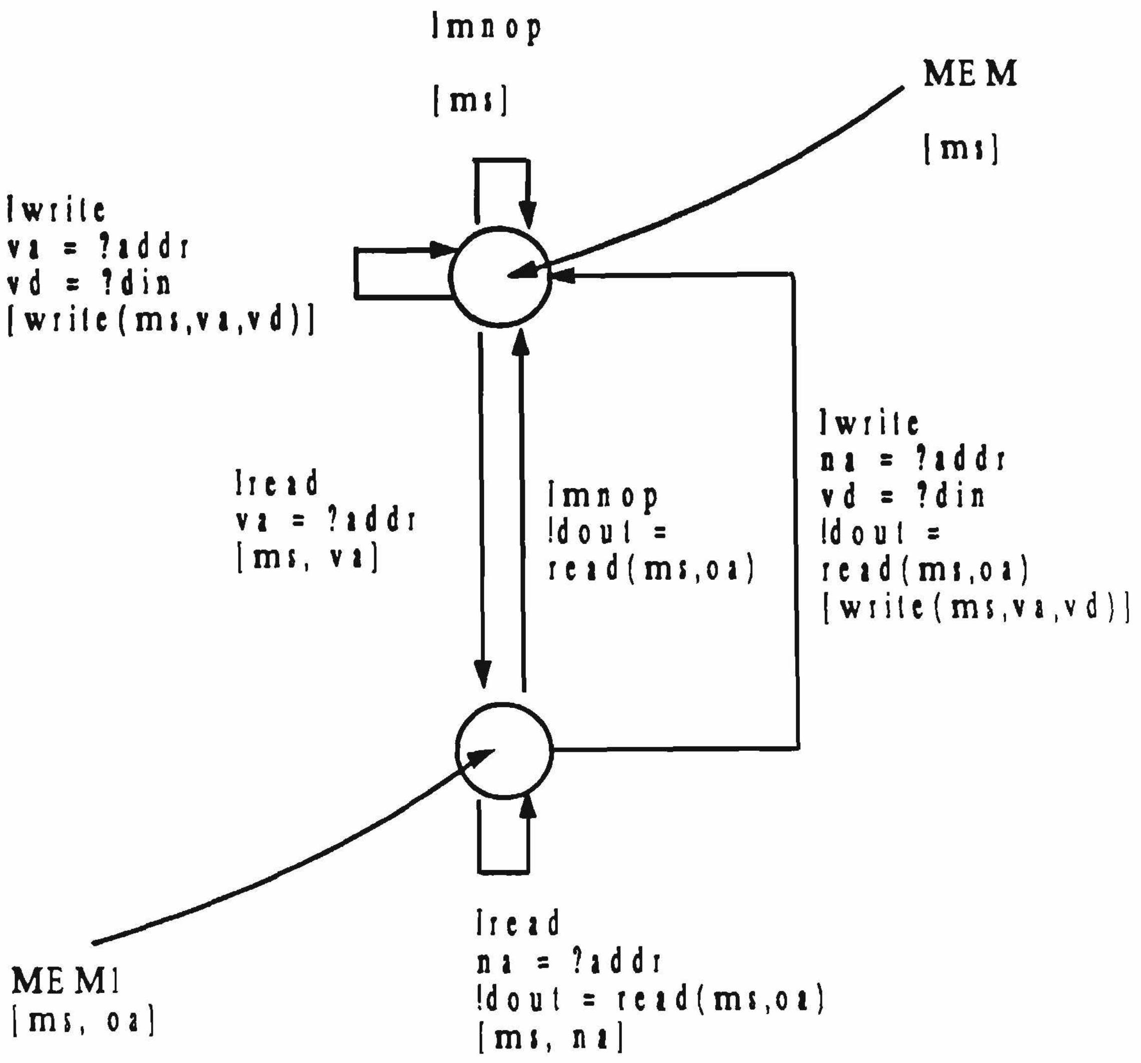


Figure 6: Depiction of the PROTOCOL Specification of MEM

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 [GS88] 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 from outside, 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`.

In this specification, the user can model datapath states using an abstract data type of his/her choice. The unit of time is unspecified. 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 `Inop` 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 [Mil80]. 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.

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 10. (Again, 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 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.

Now 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`.)

2.3 Links to Formal Verification

PARCOMP can be used to simplify the task of verifying hardware realizations. Suppose that the designer had written an independent ABSPROC specification for the stack, as shown in figure 11. Here, the operator `~>` signifies "indefinite delay"— the designer doesn't know the exact timings. The designer also uses an Abstract Data Type "stackType" to model the data path state. We can then prove that `STKREQ` and `STACK` are indistinguishable with respect to the set of commands they can perform and the results they would deliver. We omit further details.

```
--------------------- An up/down Counter ---------------------

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

--------------------- A stack controller ---------------------

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: the ''nop'' events have to be specified at present.
-- Could be specified as defaults later.
```

Figure 7: Specifications of the Submodules of the stack

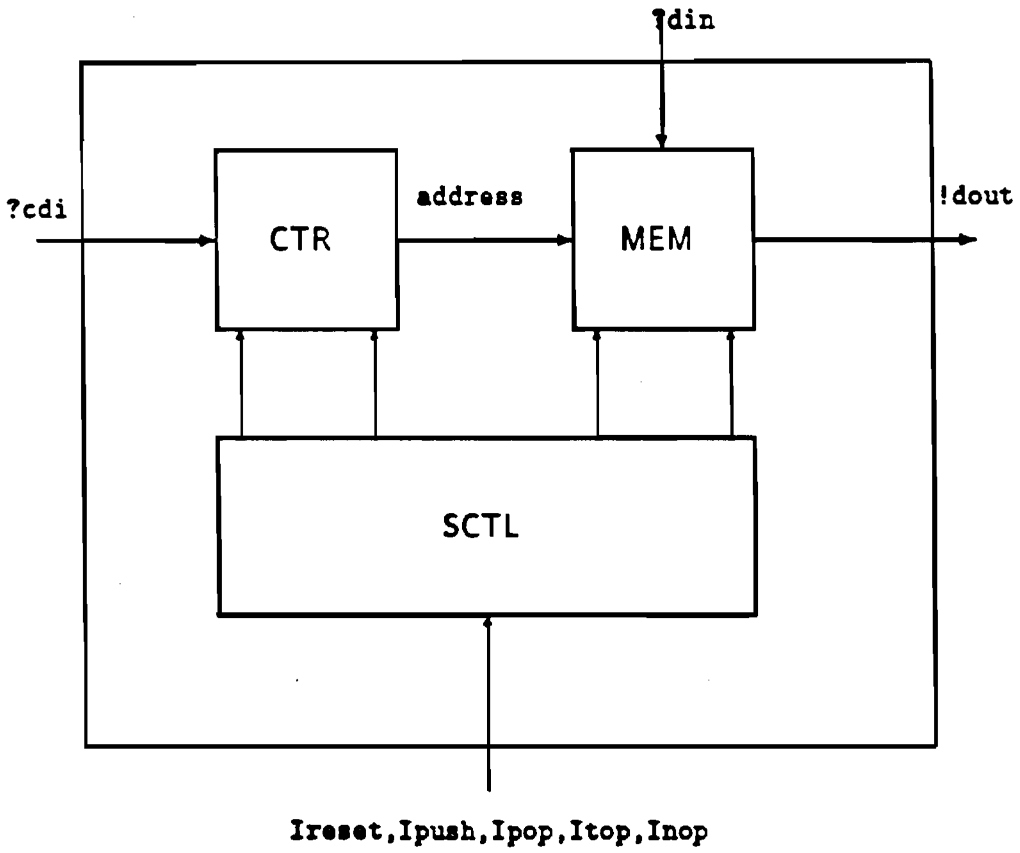


Figure 8: Schematic of the Realproc of a Stack

```
REALPROC stack [<various size & type parameters>]
PORT
  ?cdi, ?din, !dout : <suitable types>
EVENT
  Ireset, Ipush, Ipop, Itop, 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

```
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 10: Absproc Automatically Inferred from stkreal using PARCOMP

```
STKREQ [dps : stackType] <=
   Ireset ~> Ofree -> STKREQ[reset(dps)]
 | Ipush  ~> Idata_avail, vdin = ?din ~> Ofree -> STKREQ[push(dps,vdin)]
 | Ipop   ~> Ofree -> STKREQ[pop(dps)]
 | Itop   ~> Otop_avail, !dout=top(dps) ~> Ofree -> STKREQ[dps]
 | Isdef  ~> Ofree -> STKREQ[dps]
```

Figure 11: Requirements Specification for 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 [Plo81] 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 or as Higher-Order Logic specifications.

The operational meaning of a HOP process is its *transition relation* $\xrightarrow{ca} = Proc \times act \times Proc$ 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 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.

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

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

Figure 12: Definition of *Action Product* in HOP

example.) A complete definition of the action product operator is given in figure 12.

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, $ca1, P$ is a process which adds $ca1$ to the initials of P.

$$\textit{Add-to-initials:}\ \frac{P \xrightarrow{ca} P'}{ca1, P \xrightarrow{ca1,ca} P'}$$

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 $Oidle$.

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

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:

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

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:

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

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:

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

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

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.

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

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:

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

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:

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

Recursion

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

Indefinite Delay

The constructs introduced so far are all deterministic. However, nondeterminism seems to be unavoidable at really high-levels of specification when specifying an ordering between every event in the system can be tedious, or may not be possible. Besides this can also over-constrain the specification, thus leading to inefficient hardware designs.

We have begun using an "indefinite delay" operator for writing a priori specifications. The phrase "$\leadsto$ e" stands for: "Delay indefinitely until e occurs." Its definition is as follows:

$$\begin{array}{lcl} P1 & \Leftarrow & ca1 \leadsto e, ca2 \rightarrow R1 \\ & \textit{is equivalent to} & \\ P1 & \Leftarrow & ca1 \rightarrow Q1 \\ Q1 & \Leftarrow & not(e) \rightarrow Q1 \\ & & \mid e, ca2 \rightarrow R1 \end{array}$$

(In the *syntactic* rules of the action product operation, a definition $not(Ie), not(Oe) \Rightarrow not(Oe)$ should be introduced. As we will show below, events can be *semantically* interpreted as propositional temporal logic variables.)

An indefinite delay before producing an output event is an instance of nondeterministic behavior. Currently we use $\leadsto$ only in writing requirements specification for a module. The corresponding implementation of the module may have any specific delay at all, corresponding to every use of the indefinite delay operator in the requirements specification.

We are working on extensions of HOP to include concurrency-related constructs, as briefly discussed in section 7.

Some Relationships of HOP with Other Languages

It is possible to view HOP as stylized formulae in either Temporal Logic or in HOL. For instance the specification in figure 13 can be modeled in temporal logic as shown in figure 14, or in Higher Order Logic [CGM86] as shown in figure 15. 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.

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

Figure 13: An Example HOP Specification

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

Figure 14: Temporal Logic Equivalent of the Example HOP Specification

$$P(s,t) \equiv Ie1 \supset (!dout(t+1) = 55) \wedge P(f(s), t+2)$$
$$\wedge \ \ Ie2 \supset (x(t) = ?din(t) \wedge Q(g(s, x(t)), t+1))$$
$$\wedge \ \ (not(Ie1) \wedge not(Ie2)) \supset \forall k \ ERROR(t+k)$$

Figure 15: HOL equivalent of the Example HOP Specification

In the HOL description, ports are treated as streams, and explicit quantification over time is used. Hiding is modeled in HOL using existential quantification, as described by Gordon in [CGM86].

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.

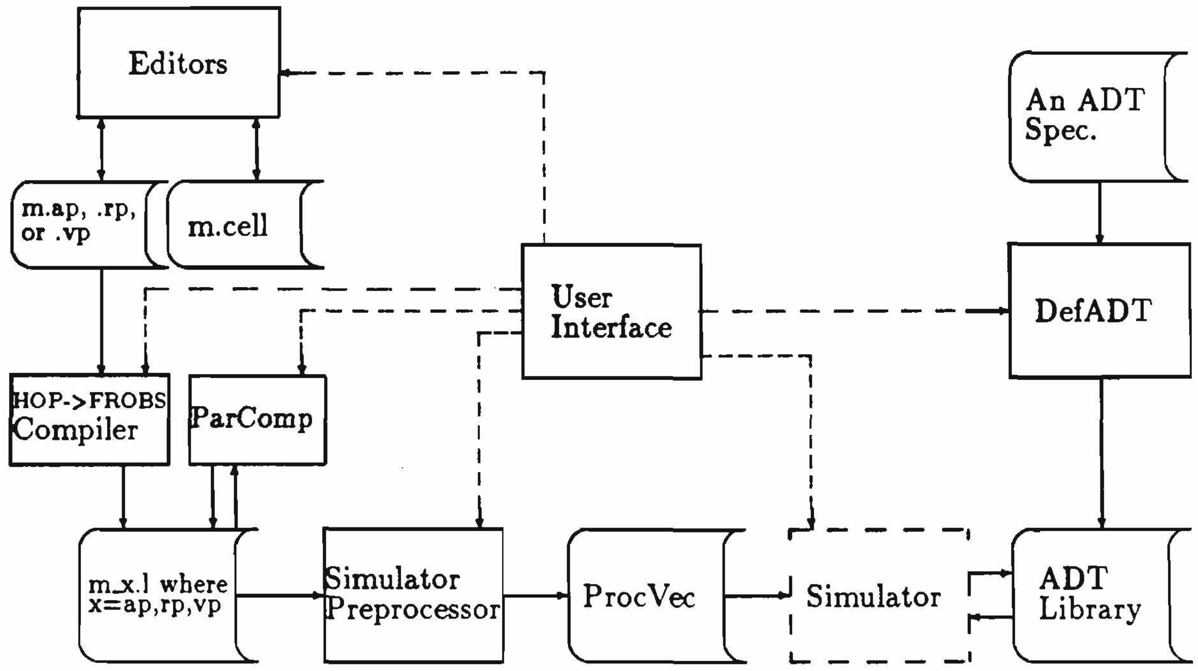


Figure 16: Data Flow Diagram of the HOP Design System

The operational semantics provided in this section define the rules of communication, and they may be understood syntactically. Milner [Mil82] and Plotkin [Plo81] 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 The HOP Design System

Figure 16 illustrates the data flow diagram of the HOP design system that is currently under development. Subsystems for which prototypes currently do not exist are shown in dashed boxes. 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.

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 layout editor called Tiler [JS86]. 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

```
STACK+TESTER [MS, CS] <= Oidle -> Oidle -> STACK+TESTER-1 [MS, 0]

STACK+TESTER-1 [MS, CS] <= Oidle -> Oidle -> STACK+TESTER-2 [MS, (Add1 0)]

STACK+TESTER-2 [MS, CS] <= Oidle -> !dot = 1
      -> STACK+TESTER-3 [(Write MS (Add1 0) 1), (Add1 0)]

STACK+TESTER-3 [MS, CS] <=
      Oidle -> STACK+TESTER-4 [(Write MS (Add1 0) 1), (Add1 (Add1 0))]

STACK+TESTER-4 [MS, CS] <= Oidle -> !dot = 2
      -> STACK+TESTER-5 [(write (Write MS (Add1 0) 1) (Add1 (Add1 0)) 2),
                         (Add1 (Add1 0))]

STACK+TESTER-5 [MS, CS] <= Oidle -> Oidle
      -> STACK+TESTER-6 [(write (Write MS (Add1 0) 1) (Add1 (Add1 0)) 2),
                         (Sub1 (Add1 (Add1 0)))]

STACK+TESTER-6 [MS, CS] <= Oidle -> Oidle ->
  !result =
   (READ
    (WRITE (WRITE MS (Add1 0) 1) (Add1 (Add1 0)) 2)
    (Sub1 (Add1 (Add1 0)))
   )
                    -> STACK+TESTER
```

Figure 18: Inferred Behavior of the Stack Interacting with the Tester

```
STACK [MS,cs]
 <=  Itop -> STOP [MS,CS]
   | Ipop  ... same as before ...
   | Ipush ... same as before ...
   | Isnop ... same as before ...

** Protocol Error in Input Specifications **
```

Figure 19: Inferred Behavior of the Stack using an Erroneous SCTL

```
TESTER <=  Oreset -> !cot = 0
                  -> Opush  -> Oidle -> !dot = 1
                  -> Opush  -> Oidle -> !dot = 2
                  -> Opop   -> Oidle
                  -> Otop   -> Oidle -> topval = ?dit, !result = topval
                  -> TESTER
```

Figure 17: Description of the Stack Tester Process

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 [MLG]. 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.

We now list specific results obtained to date. Details have been omitted.

- The stack realproc was subject to PARCOMP. The result is shown in figure 10. Considerable pruning of the state space was obtained by capitalizing on the *event hiding* information. See appendix 8.1 for details of PARCOMP.

- We deliberately introduced mistakes into the stack controller. Here is a specific experiment: do not generate the `Oread` event after synchronizing on event `Itop`, in process SCTL. PARCOMP is able to detect this as an error.

  This is possible because at this point in time, the MEM process offers a choice of input events, none of which is asserted by any other process. Thus, the behavior of the MEM process, and hence the stack process beyond this point is undefined. See figure 19 which shows the STACK process entering a state called STOP. 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.

- We composed the stack with a *tester process*, and deduced the behavior of the "tester+testee" system using PARCOMP. The particular tester we used, and the results we obtained are shown in figure 18. We now explain this result in some depth.

Figure 17 shows the stack tester designed by the designer to test the stack for the following sequence of operations: apply reset; then push 1 into the stack; then push 2; then pop the stack, and finally observe the top of the stack and return it through the !result port. In order for this experiment to succeed, the design of the stack has to be correct, and more importantly, the timing protocol followed by the tester in using the stack should also be correct! The result of PARCOMP shown in figure 18 shows that both are correct in this case.

As a specific example, the final result delivered has been inferred completely symbolically to be:

```
!result =
```

$$\begin{tikzcd}[column sep=huge, row sep=huge] H1 \, , \, H2 \arrow[r, "\parallel"] \arrow[d, dash, dashed] & H \arrow[d, dash, dashed, "\textit{(Dashes indicate Equivalence Under Functional Simulation.)}"'] \\ C1 \, , \, C2 \arrow[r, "\text{Connect}"] & C \end{tikzcd}$$

Figure 20: HOP Provides a Compositional Model

```
(READ
 (WRITE (WRITE MS (ADD1 0) 1) (ADD1 (ADD1 0)) 2)
 (SUB1 (ADD1 (ADD1 0)))
)
```

From this inferred behavior it is clear that all value communications that would occur between the various modules have been "compiled away"—i.e. statically determined via symbolic simulation, and are represented as function calls. Since function calls are much cheaper than maintaining and propagating node values, simulation can be greatly speeded-up.

Further, it can be noted that many of the functional expressions contained in the inferred behavior can be simplified using simple rewrite rules. For example, Add1 and Sub1 cancel.

This optimization needs to be implemented in our system. Since FROBS comes with a forward-chaining inference engine, and since the ADT library is implemented using FROBS, the implementation of the expression simplifier will be very modular in the HOP design system.

5 From HOP Specifications to VLSI Layout

In this section, we present a VLSI design technique that is supported by HOP. In this approach, the circuit/layout level is realized using PPL (path programmable logic). (The use of a high-level language for PPL has previously been studied by [SR85].)

PPL is a cellular grid-based design methodology. The layout is accomplished by tiling a plane with predefined cells, called the PPL cells. Until recently, one disadvantage of PPL was the limited set of predefined cells; however the capability for users to add (compact) custom cells is planned. Considering this, HOP seems to be an attractive PPL cell library specification language.

Figure 20 illustrates the correspondence we will aim for between the PPL and HOP levels; if this correspondence were to be established in some rigorous way, it could pave the way for the development of provably correct circuits from verified HOP specifications. PARCOMP could then serve as a "high-level circuit extractor."

For illustration we show how a D flip-flop fed through a tri-state driver can be described both in PPL and HOP. Figure 21 shows the PPL circuit schematic of the

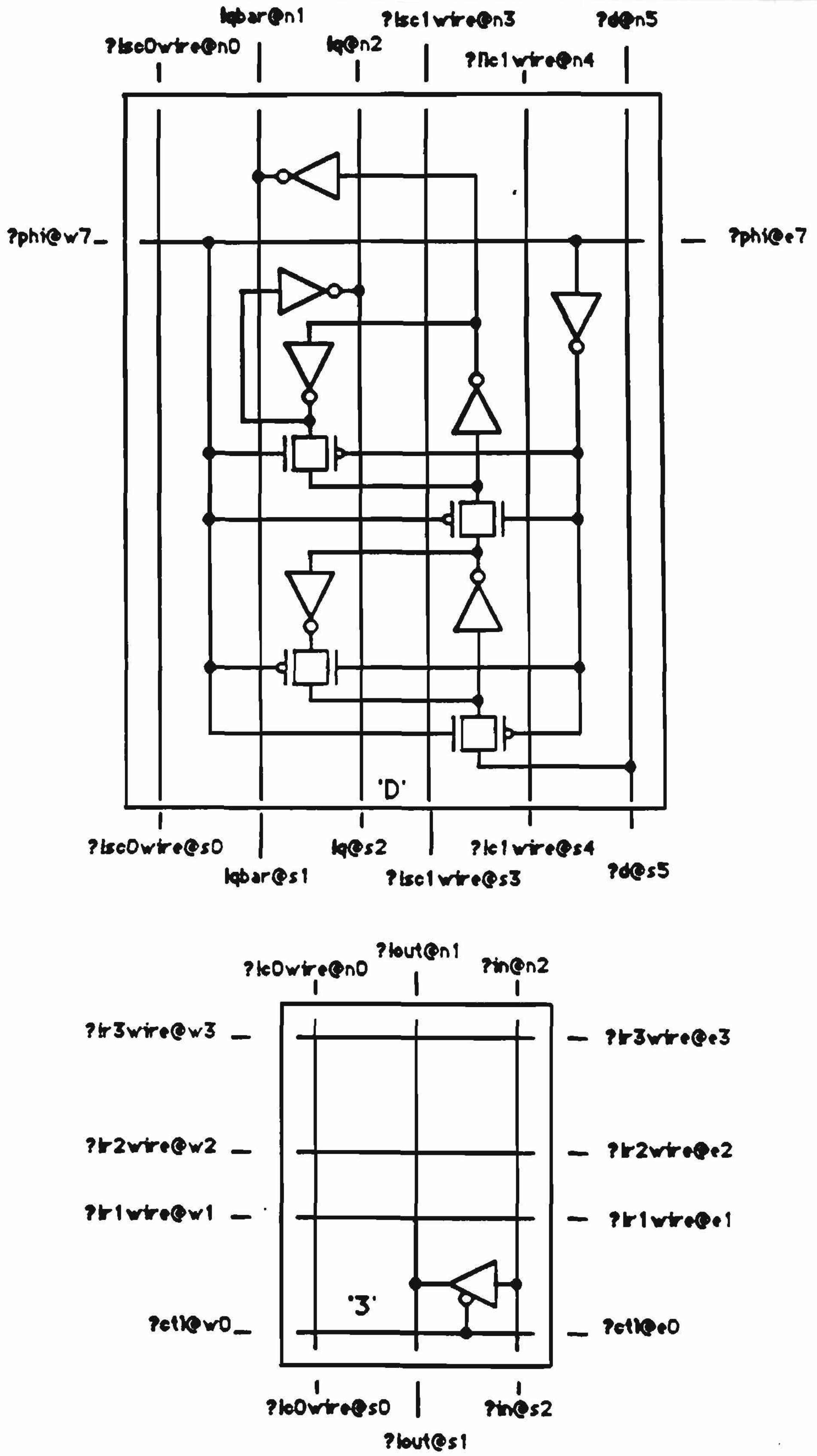
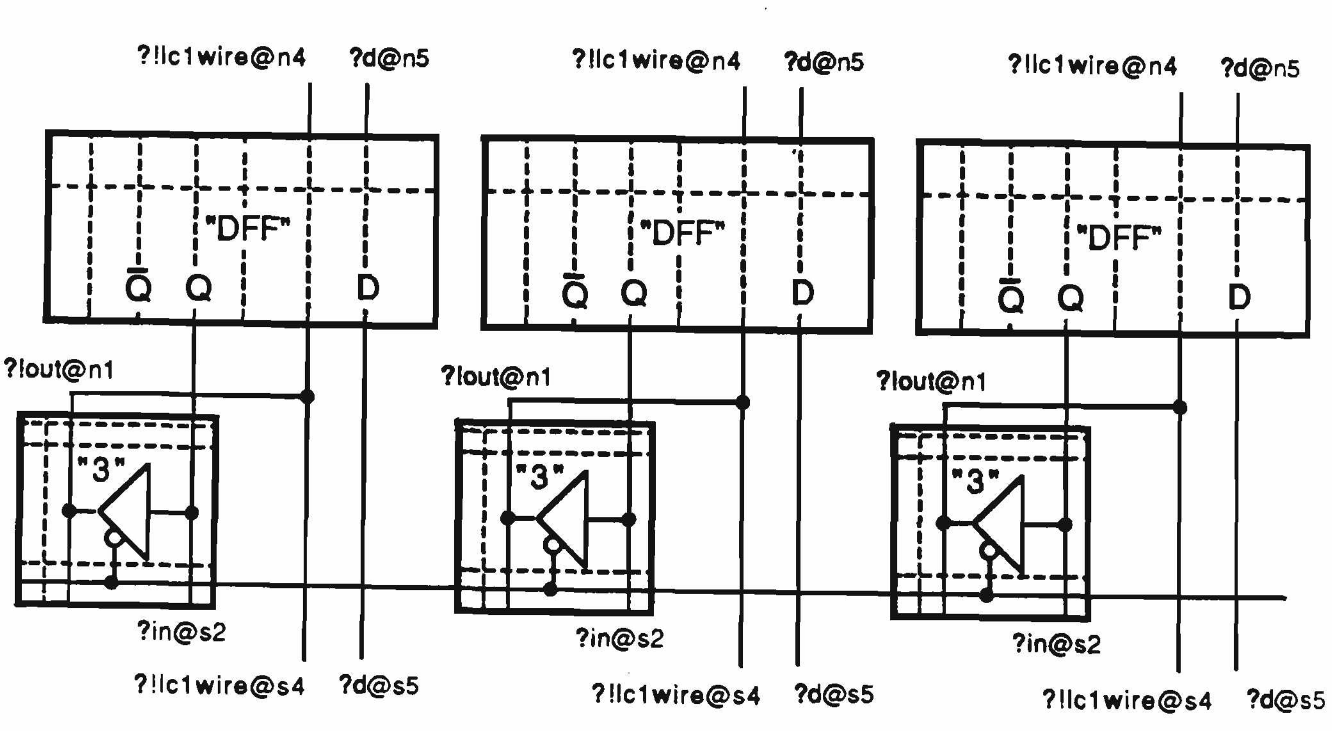


Figure 21: The Circuit Schematic of a 'D' cell Connected to a '3' cell

```
ABSPROC dff[] -- a D flip flop
PORT
   ?phi@e7, ?phi@w7, ?d@s5, ?d@n5, !q@n2, !q@s2, !qbar@n1, !qbar@s1,
   ?!sc0wire@n0, ?!sc0wire@s0, ?!sc1wire@n3, ?!sc1wire@n3,
   ?!c1wire@n4,  ?!c1wire@s4 : bit
CLOCK singlephase ?phi
EVENT
  Iload = (?phi)
  Ihold = (bit-not ?phi)
PROTOCOL
  dff[dps1,dps2] <=
            Iload, vd=?d, !q = dps2, !qbar = not(dps2) -> dff[not(vd), dps2 ]
         | Ihold, !q = not(dps1), !qbar = dps1 -> dff[dps1, not(dps1)]
END dff

-----------------------------------------------------------------

ABSPROC 3 [] -- a tristate driver
PORT
  ?in@s2, ?in@n2, ?!out@s1, ?!out@n1, ?ctl@wo, ?ctl@e0,
  ?!r2wire@e2, ?!r2wire@w2, ?!r3wire@e3, ?!r3wire@w3,
  ?!r4wire@e4, ?!r4wire@w4, ?!c0wire@n0, ?!c0wire@s0 : bit

PROTOCOL
  !out (if ?ctl ?in Z)

END 3
```

Figure 22: HOP Specifications of the 'D' and '3' cells



```
                         1 1 1 1 1 1 1 1 1 1 2 2
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
12:
11:   |" " " " " " " " " " " " " " " " " " " "
10:   |" "|" "|" "|" "|" "|" "|" "|" "|" "|" "|
 9:   |" "|" "|" "|" "|" "|" "|" "|" "|" "|" "|
 8:   |D "|D "|D "|D "|D "|D "|D "|D "|D "|D "|
 7:   |#   #
 6:           |#   #
 5:                   |#   #
 4:                           |#   #
 3:   _   _   _   _   _   _   _   _   |#  _#
 2:   |* *|* *|* *|* *|* *|* *|* *|* *|* *|* *|
 1:    "   "   "   "   "   "   "   "   "   "   |
 0:   |3  _3  _3  _3  _3  _3  _3  _3  _3  _3 _
```

Figure 23: An array of D-flip flops and tri-state drivers in PPL

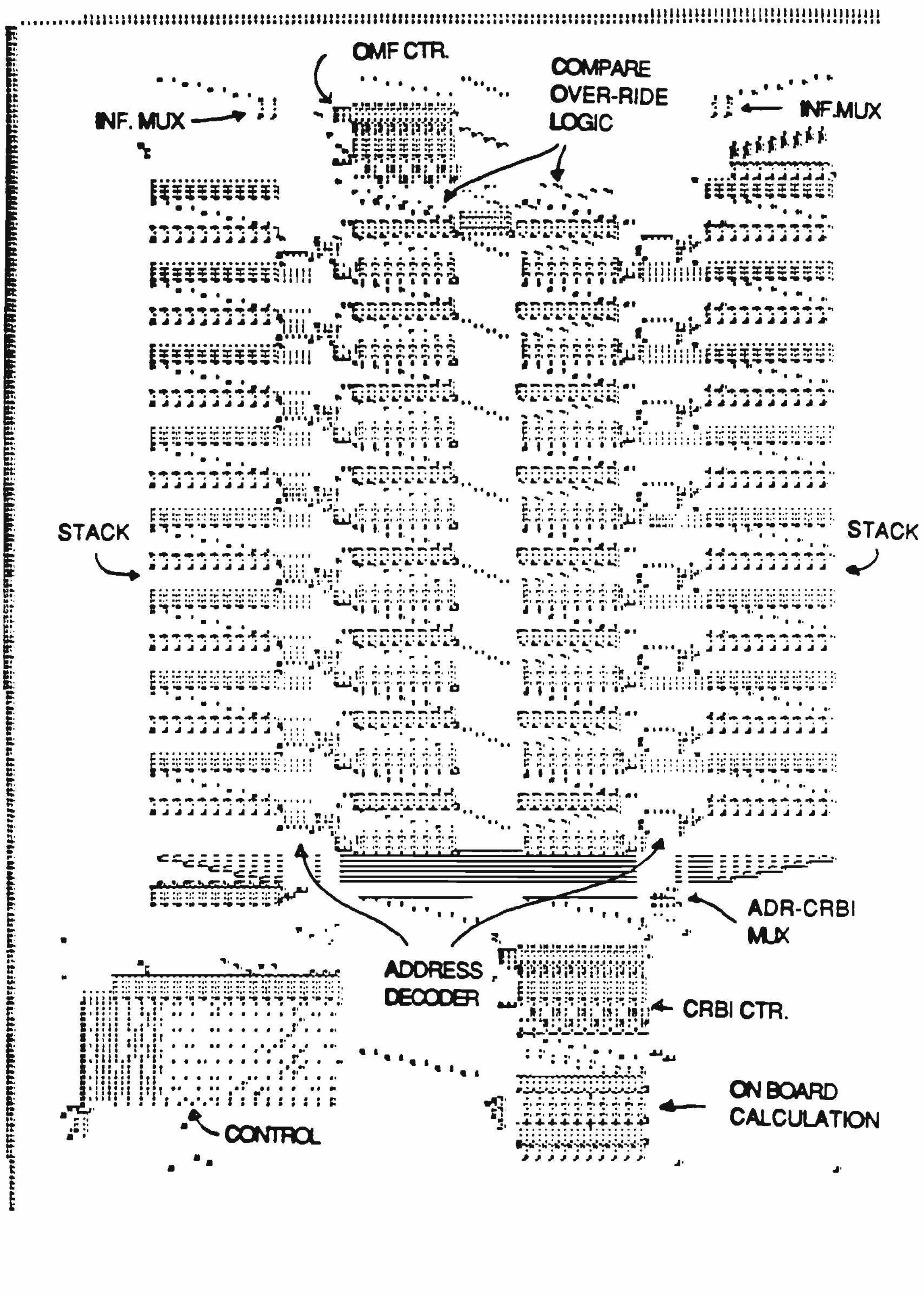


Figure 24: The Layout of the RBHC Chip in PPL

"3" cell (a tristate driver) and the "D" cell (a D flip-flop). Shown in figure 22 are the HOP descriptions for these cells. The HOP description gives a physical description of the location of ports along with a behavioral description of the cell. For example, the port labeled "?ctl@w0" in the '3' cell corresponds to the an input port named "ctl" located at the west side of the zero'th row location. All of the port numbers are relative to the lower left hand corner of the cell. In PPL the connections between ports are implied based on their placement in the circuit plane.

Figure 23 shows a PPL circuit as viewed when tiling a circuit using PPL tools. Such an ASCII file is all that is needed to generate the CIF code and have the chip fabricated (of course there are no pads shown in this figure, but they could be easily placed). This figure is an excerpt from the design of a large chip called the Rollback History Chip (RBHC) which we have conducted. The corresponding schematic is also shown in the same figure. PARCOMP could be run on the realproc generated from the PPL description, thus giving a new behavioral and physical description of the circuit. PARCOMP-DC (section 8.2) also seems promising since PPL systems often possess high degrees of geometrical regularity. As can be seen, layout composition is paralleled by PARCOMP at the HOP level. This process could be repeated to any level of design. Simulation can then be performed on the realproc with much improved speed. Verification can also be performed by the designer.

To illustrate that this technique can be used even for non-trivial designs, we present the complete layout of the RBHC in figure 24.

6 The Design of the Roll Back Chip (RBC)

In this section we present a case study of specification driven design. The example considered is called the Roll Back Chip, which forms part of the Utah Simulation Engine. Our presentation will be qualitative in nature, and is intended to highlight the issues that arise during the design of a large hardware system. We present four levels of refinement of the RBC.

Introduction to the RBC

Simulation plays an important role in the study of systems. For instance, a proposed computer architecture must be evaluated through simulation before it is built; an assembly-line simulated before it is put into operation. Such simulations are usually conducted using *discrete event simulation* techniques. Processes in the real-world are modeled as logical processes in the simulator. These processes communicate using time-stamped messages. A (fairly strong) sufficient condition for correctness in discrete event simulation is that messages be processed in non-decreasing time-stamp order, thus preserving data dependency relations.

Due to the decreasing cost of computers, there has been growing interest in speeding up simulation by spreading the logical processes of a simulation in the various nodes of a multiprocessor. A central global clock cannot be used if the simulation is asynchronous, forcing one to use independent, local clocks. Unfortunately many strategies that have been proposed for doing so are fraught with problems such as the proliferation of null messages, the risk of running into a deadlock, etc. [Fuj88]. Time-Warp is a promising alternative that doesn't suffer from the problems due to null messages or deadlock. To support time-warp, the following capabilities are necessary:

- It must be possible to take a snapshot of a process's data segment at any instance of time, and save this snapshot.
- It must be possible to restore a saved snapshot; by restoring a snapshot we mean throwing away the current data segment and reverting to a previously saved data segment.

Doing the above operations in software has proven to be extremely expensive. On the other hand it has also been observed that if the above operations could be made negligibly expensive, then time-warp would be one of the best possible approaches to distributed discrete event simulation.

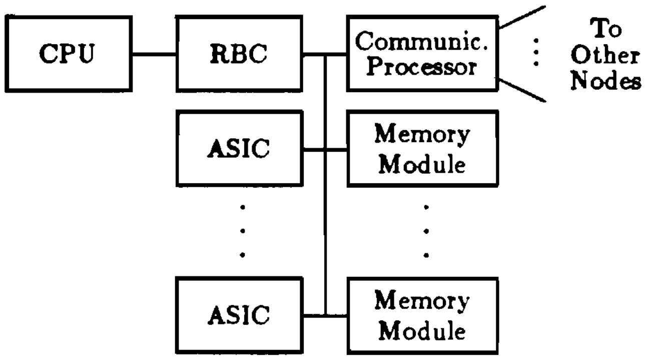


Figure 25: Configuration for each node of the Utah Simulation Engine

At the University of Utah, a hardware architecture called the USE (Utah Simulation Engine) has been under development for the past year for speeding up distributed discrete event simulation using time-warp. The Roll Back Chip (RBC) is a key component in USE. A node of the USE is shown in figure 25.

In this section we present the development of the RBC to date. The vesion of RBC presented in this paper corresponds to that presented in [FTG88]. Chronologically, the development of the RBC progressed along the following lines:

- We studied the problem at hand and captured the behavior of the intended architecture in pseudo-code;
- We specified the early design of the RBC in HOP. In this specification we used a data type similar to an ordinary stack to model the data path state of the RBC (discussed in section 6.1). This reference specification has proven to be valuable in many ways (to be discussed).
- We identified heuristics for speeding-up the initial design specification by identifying "clever data structures and algorithms"—i.e. by adopting problem-specific heuristics;
- For each such heuristic, we informally argued why each optimization strategy adopted is sound;
- We evaluated the architecture by writing a simulation of the chip in the C programming language. (The implementation of the HOP simulator isn't complete yet.)
- We coded the original reference specification in C and compared the behavior of the detailed description of the RBC against the reference specification via simulation. Both systems were subject to over a million operations, and the

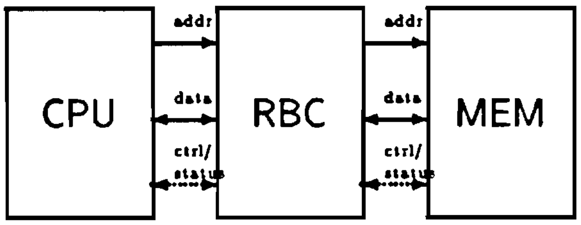


Figure 26: The Architectural Context of the RBC Chip

results produced were compared. (A form of "bisimulation".) Though not as effective as formal verification, we did discover a few bugs via this process. This supports the observation that formal reference specification directed simulation is useful in practice.

- A key component of the Roll Back Chip, the "Roll Back Cache" (RBCache) was designed, laid out, and simulated using the Path Programmable Logic (PPL) Tools [NM]. A complete specification of the TLB in HOP is currently being written. Currently another major component, the "Roll Back History Unit" has been fully designed and simulated in parts, and is being laid out in PPL.

The purpose of the remainder of this section is to reenact the development of the RBC (Parnas calls this "faking a rational design process" [PC86]). It has been our experience (hopefully shared by many of you) that as opposed to the development of relatively well-known architectures such as microprocessors, etc. the development of large, non-conventional VLSI architectures involves several iterations of the design, adjusting it to newly discovered opportunities as well as newly discovered hindrances almost on a day-to-day basis. Although one hopes that such changes be discovered as well as performed automatically on a formal specification of the system without human involvement, a more tangible hope seems to be that formal specifications be used in the following roles:

- Provide a notation for the digital system architect that is *formal*, *easily amenable to verification*, and *practically oriented*;
- Support the recording of "hunches" leading to optimizations as *invariants, or lemmas* to be proved subsequently; The LP theorem prover [GG88] can be used for reasoning about such equational formulae. (See also [GGS88].)
- Support the evaluation of performance (speed etc.) of the design using simulation studies; conduct only informal design verification at this stage, as the proposed design may not be selected owing to poor performance;
- Conduct a formal verification of the entire system when the final architecture has been selected.

Some of the levels of refinement of the RBC system are now examined.

6.1 The First Level: System RM1

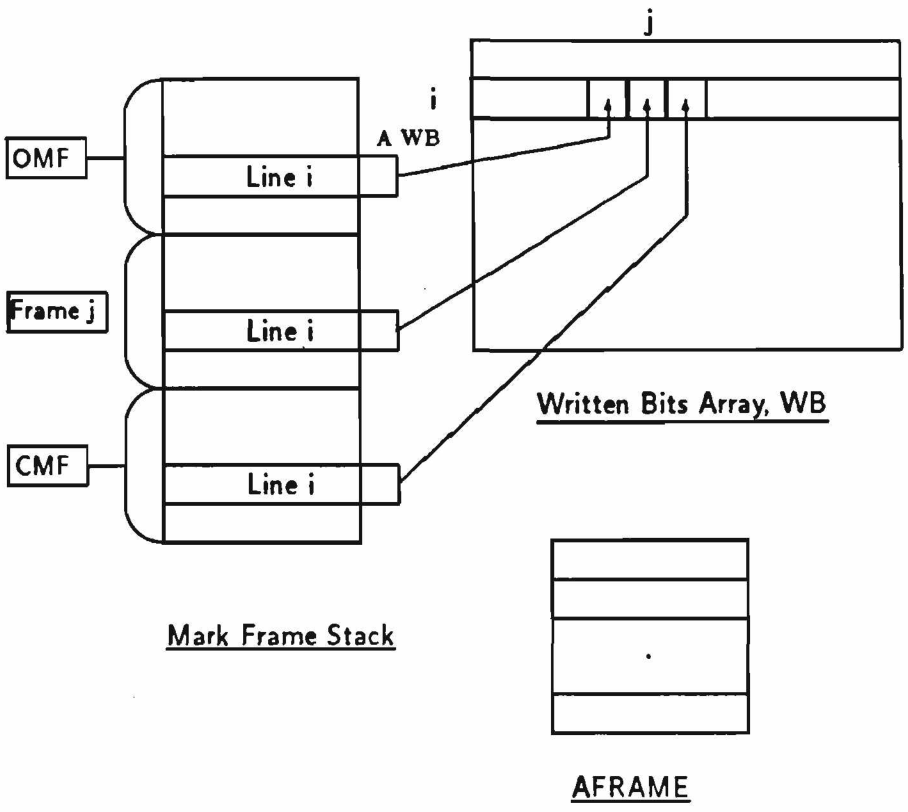


Figure 27: The First Two Levels: Systems RM1 and RM2

One way to understand the RBC is as a memory management unit situated between the CPU and the memory, as shown in figure 26. The behavior of the RBC plus a memory module from the point of view of the CPU is now described (Note: the name "RM1" corresponds to RBC plus Memory, level 1.)

- A program P running in the above system uses a data segment ranging from address *Amin* to *Amax*. This is called the *version controlled memory* of which snapshots can be taken and restored.
- At any time P can *read* from the data segment from address *A* (operation *Read*(*A*)).
- At any time program P can *write* into the data segment at address *A* the data *d*.
- Periodically, program P issues a request to the 'RM1' system to take a version ("snap-shot") of the data segment contents; P then continues with the computation; this operation is called *Mark*().
- At some future time, P encounters an error and wants to revert to an earlier version of the data segment state, effectively restarting the computation at some time in the past. This operation is called *Rollback*.
- At any time P can discard very old versions of checkpointed state that are no longer needed. This operation is called *Advance*.

Figure 27 shows a data structure similar to an ordinary stack that was used to model the above requirements specification. (Please ignore the WB array and the Aframe for now.) Specifically, the data type used is:

```
type RM1type =
  RECORD
    STK : array [nframes] of array [nlines] of <data,wb>
    CMF, OMF : int
  END
```

The data type consists of a stack STK and two pointers CMF and OMF. STK contains `nframes` frames, each of which is as large as the data segment. STK is used to hold the multiple snapshots of the data segment. CMF points to the current "mark frame", or the current data segment. OMF points to the oldest mark frame in use. Each frame of the stack holds `nlines` "lines" of data. A line is a group of bytes, and is similar to the lines in a cache. Actually pairs `<data,wb>` are maintained. The `wb` stands for "written bit", and is set only if the line has been written into. A `wb=0` corresponds to a "hole"; *e.g.* if location i of a frame F has a zero written bit, then location i was not written into while F was the "current" mark frame.

(For simplicity assume that the fields of the record correspond to variables with the same names as the fields.) The operations of the RBC are:

read(a) finds the least f starting from $f = CMF$ such that STK[f][a].wb is set; it then returns the data item STK[f][a].data. Frame f in this example corresponds to the *most recent version* (MRV) of the data at line address a. Hence we will call such a frame the "MRV frame".

write(a, d) updates STK[CMF][a].data with d, and also sets STK[CMF][a].wb.

mark(k) increments CMF by k.

rollback(k) decrements CMF by k, and if CMF falls behind OMF, an error is reported.

advance does nothing at all—we are not interested in garbage collecting the conceptual stack!

6.2 The Second Level: System RM2

We move towards practical reality by making the following changes (see figure 27):

- We detach the written bits from the stack and pool them into a matrix;
- We treat the stack as a circular buffer; the data area is shown side-by-side with the WB array;
- We make a provision for garbage collection by introducing the frame called Aframe. The idea behind garbage collection is simple. When the OMF frame is no longer needed (this happens when the *Global Virtual Time*(GVT) [Jef85] exceeds the the time-stamp of the OMF frame), we may free-up the OMF frame *provided* we are not throwing away a legal version of any data. The steps in archiving are the following:

```
Suppose the GVT advances to frame OMF+K. Then
For every line l that doesn't have a set written bit in frame OMF+k
  determine the MRV of the data looking back from OMF+K;
  copy this MRV data to Aframe[l]
end For;
Free up frames from OMF to OMF+k.
```

All the operations in level RM2 are implemented as in level RM1 except for the MRV computation. In RM2, if no set written bits exist for a line l between positions CMF and OMF, then the MRV is read from Aframe[l].

From the point of view of the user, all the levels of the RM system are best regarded as abstract data types with *constructors write*, *mark*, *rollback*, and *advance*, and the only observer *read*. Hence the verification condition for *functional correctness* can be stated using *data type induction* [GHM78]. For example, the verification condition going from level RM1 to level RM2 is:

For all constructor operations *op* defined at level RM1 (op_{RM1}) and RM2 (op_{RM2}) and their arguments *oparg*

$$\frac{read_{RM1}(dps_{RM1}, l) = read_{RM2}(dps_{RM2}, l)}{read_{RM1}(op_{RM1}(dps_{RM1}, opargs), l) = read_{RM2}(op_{RM2}(dps_{RM2}, opargs), l)}.$$

Each level following RM2 has to be similarly verified with respect to RM1.

For an earlier design of the RBC chip, we did prove some of these invariants by hand by unfolding the consequent part (the portion below the line) using the definition of *read* and *op* at the lower level, performing a case analysis on it, and showing that in each of these cases the consequent either reduces to an instance of the antecedent or follows from the antecedent. In a few cases, proof by contradiction proved to be much easier.

Our current focus is in formulating verification criteria, and understanding what exactly verification means in the context of complex timings.

6.3 The Third Level: System RM3

Obviously, searching down the stack for every *read* operation is inefficient. Hence we introduce a cache unit to cache the data in the MRV frame (see figure 28). The management of this cache is quite different from traditional cache units because we have to guarantee that whenever there is a cache hit, we do return the MRV data item. This also means that when a roll-back or advance occurs, suitable invalidations of the cache entries are performed.

At level RM3 we have three processes, LRU, RBCache, and CTRLR communicating via their interface protocols. The process CTRLR may not realized as a stand-alone unit; it simply represents the portion of the microcode devoted to the management of level RM3.

It is possible to model the system state as a three tuple of the states of these modules and specify the RBC operations as mappings on these states. A more useful approach—the one we plan to use—would be to model these as stand-alone processes, compose them using PARCOMP, and then compare the process inferred by PARCOMP with a process representation of level RM2. While we have begun working in this direction, we do not have a satisfactory solution yet. The questions we would like to answer are: (i) how best to write a prior specifications in a manner that permits systematic refinement to more detailed timing? (ii) how to discover optimizations such as pipelining in this process? (ii) What forms of nondeterminism are useful at the architectural level of specification?

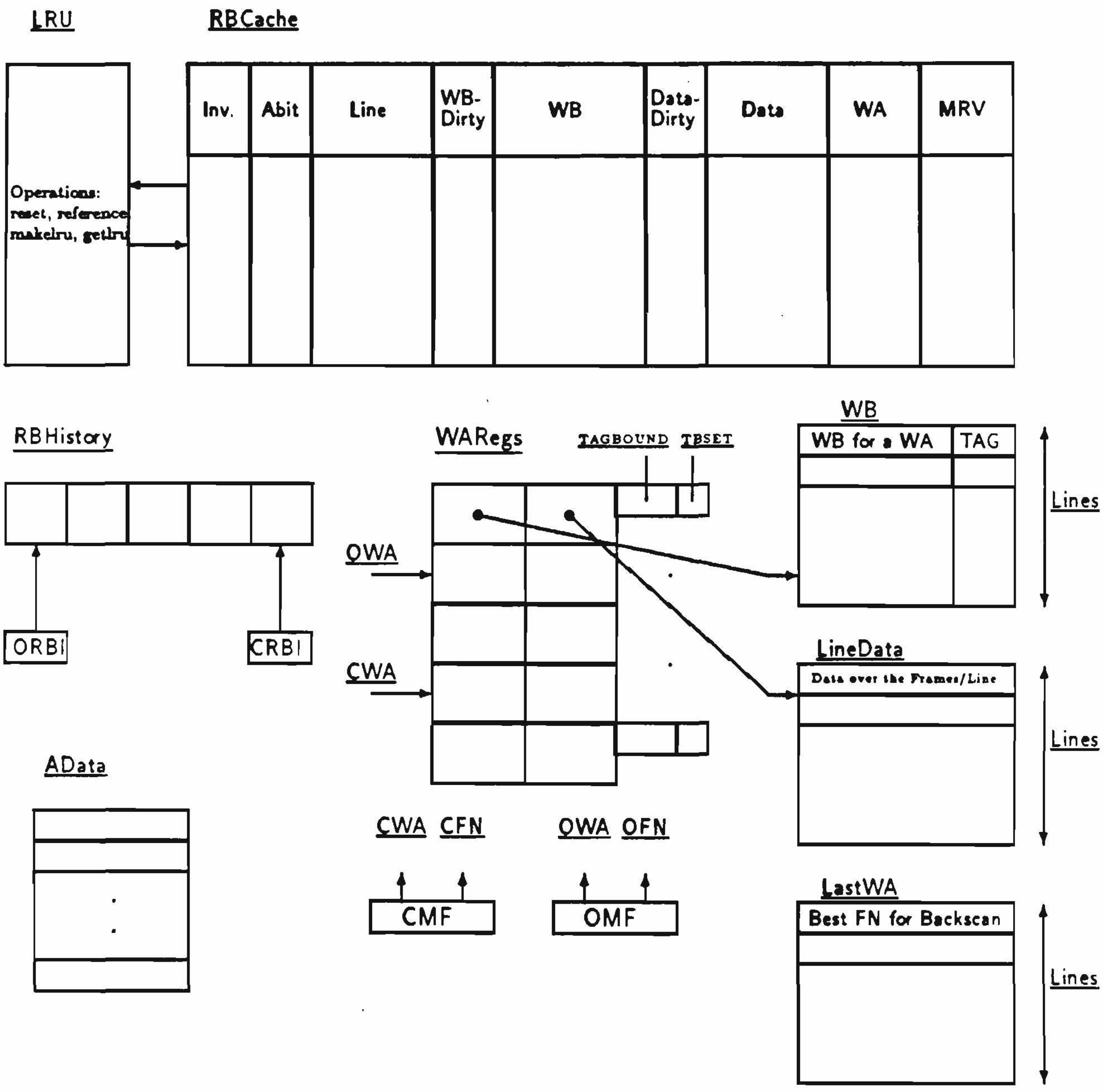


Figure 28: The Levels RM3, RM4 and RM5

6.4 The Fourth Level: System RM4

When a cache miss occurs in the RBCache, the MRV entry has to be retrieved from the memory. A backward search beginning at CMF searching for the first set written bit has to be performed. Is it possible to keep a good estimate of the point from which to begin the backwards scan? The answer is 'yes', and incorporating this optimization leads to model RM4.

In this scheme (shown in figure 28), a frame full of pointers (one for each line) is maintained in the array LastWa. The contents of LastWa are updated *whenever a cache entry is evicted.* Our simulation studies show that the use of LastWa cuts down the average distance of backscan for the MRV dramatically.

6.5 The Fifth Level: System RM5

When a Rollback occurs, not only must we invalidate entries in the cache that correspond to frames that have been "rolled over", but we must also clear the written bits in the memory corresponding to these "rolled over" frames. This is because we are effectively forgetting the existence of these frames. However, industriously clearing the written bits isn't wise. We have developed a lazy approach to clearing the written bits. In figure 28 the addition to our "data structures" for achieving this optimization is shown. The mechanism is called the "Roll Back History Unit", or the RBhistory unit.

Section 5 presents the design of the RBH unit.

Section Summary

The current design of the RBC chip is still at least five more such refinement steps away! The five steps shown so far make it amply clear to us that digital system architects routinely think of data abstractions and are aware of optimization invariants. However they don't seem to document these thoughts in a formal specification language. Here is where specification driven design approaches have a definite contribution to make. Also as we mentioned in section 1, lack of performance is also a design error. (Interestingly, the USE system could be used to conduct performance studies of digital architectures—even of itself—quite rapidly!)

7 Summary

- We have developed a hardware specification language HOP, and are using it for the specification driven design of a large custom architecture. This cooperative research is expected to lead to a language well balanced in formal details as well as pragmatic ones.
- One of the key results so far has been the recognition of the importance of data and structural abstraction mechanisms in real-world design. HOP seems to be satisfactory in these respects.
- Constructs for modeling truly concurrent processes (fork/join, barrier synchronization, mutual exclusion of resources, etc.) are missing in the present version of HOP. We are designing a successor to HOP called "HOP-CP" (CP for concurrent processes) that includes these constructs. The present version of HOP

would be truly compatible with HOP-CP. Specifications for the components of USE will be written in HOP-CP.

- A library based design approach where certified circuit/layout "tiles" are stored along with high-level specifications is believed to be an effective way to translate correctness proofs at the high-level into correctness assurances regarding the circuit/layout. Our experiments with PPL substantiate this observation.

Acknowledgements

The first author acknowledges the support of his research by the National Science Foundation under Grant No.MIP-8710874. The second author acknowledges the support of his research by ONR, under contract number 00014-87K-0184.

8 Appendix

8.1 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
$P_i[\overline{X_i}] = y_i \ : \ initials_i \rightarrow R_i(y_i);$

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

$\boxed{\textit{Output:}}$ 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:

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

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

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

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

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

$\| \{P_i[\overline{X_i}], ..., C\}$ 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

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

-

- 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 12. 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. □

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

Consider the array A shown in figure 29. It consists of a collection of modules M connected in a regular interconnection pattern. For simplicity of explanation, assume a nearest-neighbor connection that is regular in both the dimensions.

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

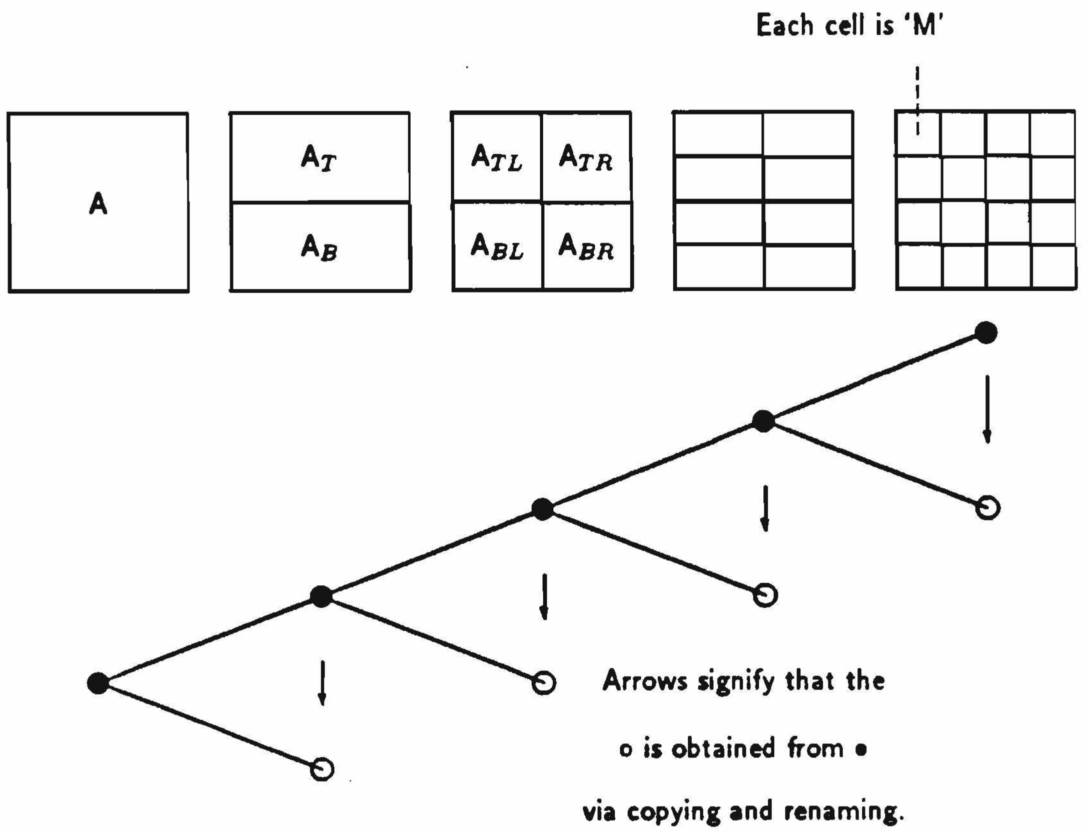


Figure 29: Divide and Conquer PARCOMP

This division process can be carried down to the leaf cells, as depicted in figure 29.

PARCOMP-DC is often more efficient than PARCOMP. Let us make an approximate cost analysis. The worst-case time complexity of PARCOMP is primarily dependent on the number of control states that we have in a process diagram. Specifically, it can be equal to the cross-product of the number of control states in each of the processes. 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

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

because we may, in the worst-case, end-up taking a full cross-product of the process diagrams of the N modules.

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 29). Then

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

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. (This is the case for our data structures that represent processes.) The number of queries and assertions in a compound action is assumed to be bounded by a constant. 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 approach is suggested for handling 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.

References

[ACFM85] 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.

[Bae86] Jean-Loup Baer. Modelling Architectural Features With Petri Nets. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 258–275, Springer Verlag, September 1986. LNCS 255.

[BCDM85] 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.

[Bro75] Frederick P. Brooks. *The Mythical Man-month.* Addison-Wesley, 1975.

[Bry84] Randall E. Bryant. A Switch Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computer*, C-33:160–177, February 1984.

[CGM86] 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.

[Chu87] 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.

[Coh88] Avra Cohn. Correctness Properties of the Viper Block Model: The Second Level. In *1988 Banff Workshop on Hardware Verification (this volume)*, Springer Verlag, 1988.

[FTG88] Richard Fujimoto, Jya-Jang Tsai, and Ganesh Gopalakrishnan. Design and Performance of Special Purpose Hardware for Time Warp. In *15th Annual International Symposium on Computer Architecture, Honolulu*, pages 401–408, 1988.

[Fuj88] R. M. Fujimoto. Performance Measurements of Distributed Simulation Programs. *1988 Society for Computer Simulation Multiconference*, feb 1988.

[GG88] Stephen J. Garland and John Guttag. Inductive Methods for Reasoning About Abstract Data Types. In *15th ACM Conference on Principles of Programming Languages*, January 1988. San Diego, CA, January 13-15.; This article describes the theory behind the Larch theorem prover (LP).

[GGS88] 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.

[GHM78] John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract Data Types and Software Validation. *Communications of the ACM*, 21(12):1048–1064, December 1978.

[Gop86] 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).

[GS88] Ganesh C. Gopalakrishnan and Mandayam K. Srivas. Implementing Functional Programs Using Mutable Abstract Data Types. *Information Processing Letters*, 26(6):277–286, January 1988.

[GSS87] 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.).

[Hen84] Matthew Hennessy. *Proving Systolic Systems Correct.* Technical Report CSR-162-84, Department of Computer Science, University of Edinburg, June 1984.

[HK87] 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.

[ISD88] 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.

[JB85] 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.

[JBB88] 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.

[Jef85] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[JS86] 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.

[Lin85] Gary Lindstrom. Functional Programming and the Logical Variable. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 266–280, January 1985.

[Mar85] 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.

[MH85] M.Lam and H.T.Kung. A Transformational Approach to Systolic System Design. *IEEE Computer*, 18(2), 1985.

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

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

[Mil83] George J. Milne. CIRCAL: A calculus for circuit description. *Integration*, (1):121–160, 1983.

[Mil85] George J. Milne. Simulation and Verification: Related Techniques for Hardware Analysis. In *Proceedings of the Seventh International Conference on Computer Hardware Description Languages*, pages 404–417, North-Holland, 1985.

[MLG] 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.

[Mue87] 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.

[NM] Mani Narayana and Surya Mantha. The Design of a TLB for the Roll Back Chip. VLSI Class Project Report, Winter 1988.

[NS88] 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.

[Pat85] Dorab Patel. nuFP: An Environment for the Multi-level Specification, Analysis and Synthesis of Hardware Algorithms. In *Proceedings of the Functional Programming and Computer Architecture Conference*, Springer-Verlag, LNCS 201, September 1985. Nancy, France.

[PC86] David Lorge Parnas and Paul C. Clements. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, February 1986.

[Plo81] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, Aarhus University, Denmark, September 1981.

[Seq87] Carlo H. Sequin. VLSI Design Strategies. In Wolfgang Fichtner and Martin Morf, editors, *VLSI CAD Tools and Applications*, pages 1–16, Kluwer Academic Press, 1987.

[She84] Mary Sheeran. muFP, a Language for VLSI Design. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 104–112, 1984.

[She85] 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.

[Sne85] Jan Snepscheut. *Trace Theory and VLSI Design.* Springer Verlag, 1985. LNCS 200.

[SR85] Pashupathy A. Subramaniam and Sanjay Rajopadhye. Formal Semantics for a Symbolic IC Design Technique: Examples and Applications. *Integration: The VLSI Journal*, (3):13–32, March 1985.

[Sto77] Joseph E. Stoy. *Denotational Semantics.* The MIT Press, 1977.

[Sub83] 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.

[WFC87] W.F.Clocksin. Logic Programming and Digital Circuit Analysis. *Journal of Logic Programming*, (4):59–82, 1987.