Simulation Effectiveness Research Report[1]

Technical Report CRI-85-29
(DISC 83-2 - April 1983)

Alice C. Parker

[1]Funding for this research was provided by IBM Corporation Grant S 956501 LX A B22.

Table of Contents

List of Figures

List of Tables

1. ABSTRACT

This report contains a summary of research performed under the general topic of "Simulation Effectiveness." The problem of design error detection in hardware is discussed, and a classification of errors is produced. Some qualitative results from a survey of designers concerning design errors are then presented. Simulation examples illustrating the errors are summarized.

Behavior Expressions are used to represent abstract behavior. Proofs are performed to show how each logical design error type in a hardware description manifests itself in the derived Behavior Expression. An extension of Behavior Expressions to represent concurrent behavior is discussed. An appendix to the report contains a paper discussing the relationship between hardware synthesis and hardware verification.

2. INTRODUCTION

Digital designs have become increasingly complex. One of the fundamental problems facing designers is that of assuring themselves that a completed design is functionally correct (outputs and their ordering are what was expected from the sequence of inputs we supplied). There are two approaches to determining the functional correctness of hardware prior to hardware construction itself:

- Simulation
- Verification

Simulation is a software tool widely used during the design process. Simulation does not guarantee that a design is functionally correct unless all valid input data combinations have been applied. It merely validates correctness on a case-by-case basis.

Verification, on the other hand, gives some confidence that a design is correct for an entire class of inputs, or for all possible inputs. However,

hardware verification is a less mature discipline than simulation; it has not been applied by industry to any large extent in a production environment.

Unfortunately, there are classes of errors which are difficult to trap by simulation. They cannot be guaranteed to be detected unless all possible combinations of input sequences are used. The consequence, exhaustive simulation, is prohibitively slow and expensive. Simulation test data must be carefully chosen in order to trap these errors, or simulation must be supplemented with other software techniques designed specifically to detect certain classes of errors.

This project has a long term goal which is to provide a method of early detection of hardware design errors, along with confidence measures regarding error coverage. In order to achieve this goal, we are currently pursuing a number of short term objectives including:

- Categorization of design errors into classes and within classes into types.
- Formal mathematical definitions of the meaning of each error type.
- Error coverage computation by type for simulation, symbolic simulation and other methods based on test set information and the formal definitions of each error type.
- Heuristics for test-set selection based on the formal definition and above error-coverage.

This report describes initial groundwork performed as an attack on this problem.

2.1. The Problem Approach

The approach to solving the error detection problem will now be outlined. The first stage in this research was to produce a clear definition of the research problem. Correct behavior had to be precisely defined, and the relationship of detection of errors by simulation to strict behavioral

correctness was described by defining simulation effectiveness.

Design errors had to be characterized by classes, and by types of errors falling into these classes. Because designers must prepare design inputs for simulation by writing (or flowcharting) the designs, we considered not only design errors per se but also the errors made while transcribing designs into formal descriptions. We did not consider errors which can be guaranteed to be detected while compiling or preprocessing the descriptions or even while performing a single simulation pass; rather we are interested in errors which might be detected at a later stage such as prototype testing. Therefore, this research focuses on errors which are timing dependent, errors which are detected only by applying specific data inputs, and errors which occur due to the concurrent nature of the hardware operation. We have divided the errors into the three broad classes

- logical errors
- timing errors
- concurrent errors.

Thus, the next step of the research has been to determine, within each of these classes, what types of design errors which are made by hardware designers are difficult to detect by conventional simulation methods. For this purpose we distributed a survey form (Design Error Survey, see Appendix A) among various design engineers aimed at pin-pointing these error types. From this survey we have obtained an idea of errors which occur frequently and which are not easily detected. In the research itself, we have dealt with each class of errors separately.

The third step has been to produce some simulation examples to illustrate error types and their manifestation in simulation outputs. These examples are described in Section 5.

A fourth step has been to define hardware correctness in a more formal manner, and to use this definition to understand the problem of error detection. Behavior Expressions are used as a way of expressing correct behavior. Proofs are performed to show how each type of logical design error in a hardware description manifests itself in the derived Behavior Expression.

The experience of the proving process is being used to investigate a practical method for guaranteeing certain levels of error coverage. As a final step, an extension to Behavior Expressions to express concurrent behavior is beginning to be investigated.

3. RELATED RESEARCH

Virtually no related research exists on test generation for simulation done to determine logical correctness. However, design errors are covered in some reliability models, and there are some analogies to software testing. With the relationship to software in mind, previous research in the areas of hardware simulation, hardware verification, software verification and software testing was examined. A computer search was conducted by searching from 1976 through 1981 on the following topics:

- Hardware Simulation
- Hardware formal specification
- Hardware timing errors
- Hardware verification
- Hardware design errors
- Hardware validation
- Hardware logical correctness
- Hardware concurrent
- Software verification
- Software test data selection

- Software correctness
- Software errors

The results of this search, along with other published research of which we are aware, will be published in a separate report.

4. ERROR CLASSIFICATION AND SURVEY RESULTS

In order to discuss design errors, first we define correct behavior or correctness of a design. We begin with a general definition of correctness and then define correctness for each class of errors separately, since the issues involved for each class of errors are different. Then, error types within each class are presented, with examples. The section concludes with a qualitative analysis of the survey results.

We define correctness as adherence of the actual behavior to a specification of behavior. An error, then, is said to exist when there is a point of disagreement between the observed behavior of the design and the specified behavior. A design is said to be correct when the adherence of the actual behavior to a theoretically complete, ideal specification is strict (point by point). Now, typical specifications are nowhere near being exact statements of the designer's intent. So, we restrict the definition of error to disagreement with the actual, incomplete specification or with the designer's intent; any behavior which is exhibited by the design but not specified explicitly by the actual specification is a "don't care" type error.

We now informally define correct behavior with respect to each class of errors.

Correct Logical Behavior:	Events have a partial ordering, events occur under certain well-defined conditions, and output values are correct with respect to input values.
Correct Timing Behavior:	Events have a partial ordering, output values are correct with respect to input values and it is possible to specify that events must

occur during well-defined time intervals under well defined conditions.

Correct Concurrent Behavior: Events have a partial ordering and occur under well defined conditions, output values are correct with respect to input values, there is an internal ordering across parallel branches and no unspecified resource conflicts[1] occur.

Within a class, errors can be restricted to a set of types. This set of types may be specified by the designer himself or may be derived automatically (from the design description). More formal definitions of behavior follow in Section 6.

Simulation effectiveness is defined as the percentage of each class of errors which can be detected by simulation, per input test set. This means that if the number of tests required to detect a fixed percentage of a certain type of error increases, simulation effectiveness decreases. Simulation effectiveness is indirectly related to the size of the test set and directly related to error coverage.

Mathematically

$$SE_{kj} = \frac{D_{kj}}{E_{kj}(\text{Card}[S_k])}$$

Where $\{S_k\}$ is a set of simulation tests for machine k, E_{kj} is the number of errors of class j which exist in the machine k, D_{kj} is the number of detected errors of class j for machine k, and Card $[S_k]$ is the number of elements in set S_k. SE_{kj} is the simulation effectiveness.

Since exhaustive testing is impractical, we define simulation correctness

---

[1]Resources include wires and interconnection logic, as well as registers and functional units.

as that which is determinable given a simulation of the design and a given test set; i.e. it is correctness relative to a test set. As simulation effectiveness increases, simulation correctness approaches correctness. Figures 4.1 and 4.2 describe some of the correlations that exist between these quantities.

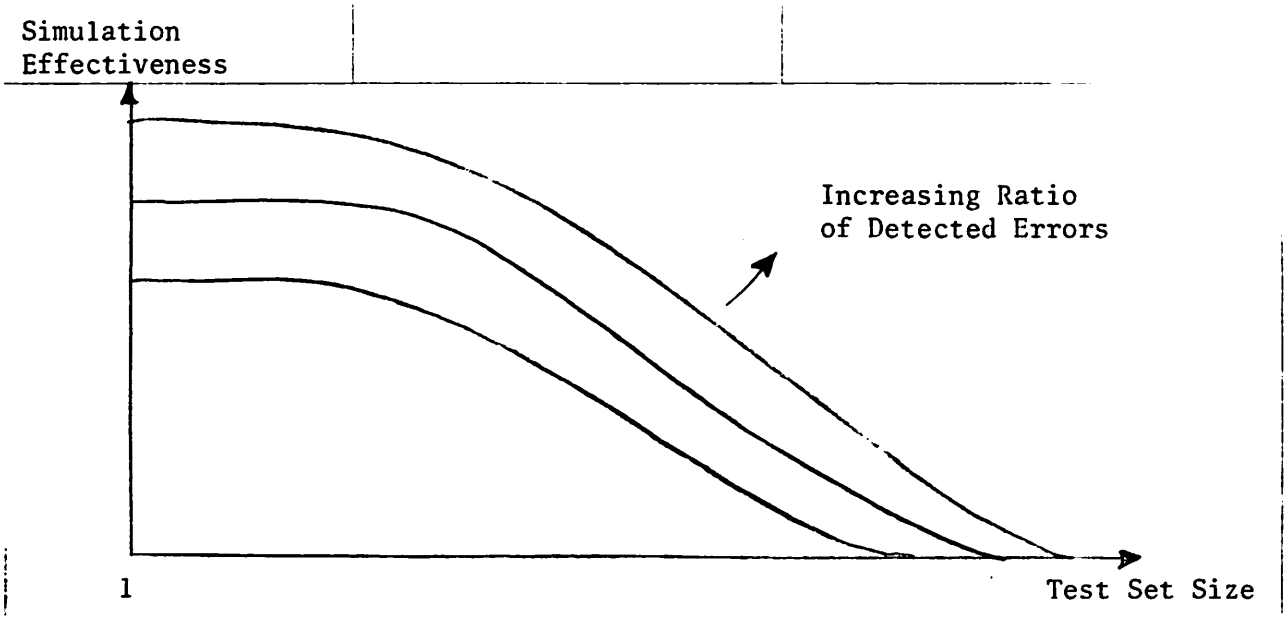


Figure 4-1: The relationship between Test-Set Size and Simulation Effectiveness

The Survey Results

In order to classify errors into types, and to determine which error types escape early detection by simulation, we conducted a survey of designers, both within and outside of IBM. A sample survey form is attached as Appendix A. Within IBM, a number of designers were surveyed, and sixty-two errors were described. Outside of IBM, designers in one company were interviewed informally, one company provided a comprehensive report on design errors, key individuals in other companies were sent survey forms for general information, and individual designers were surveyed about specific projects.

Although the survey results are incomplete, and many of the surveys have missing or unclear answers, some general trends have emerged.

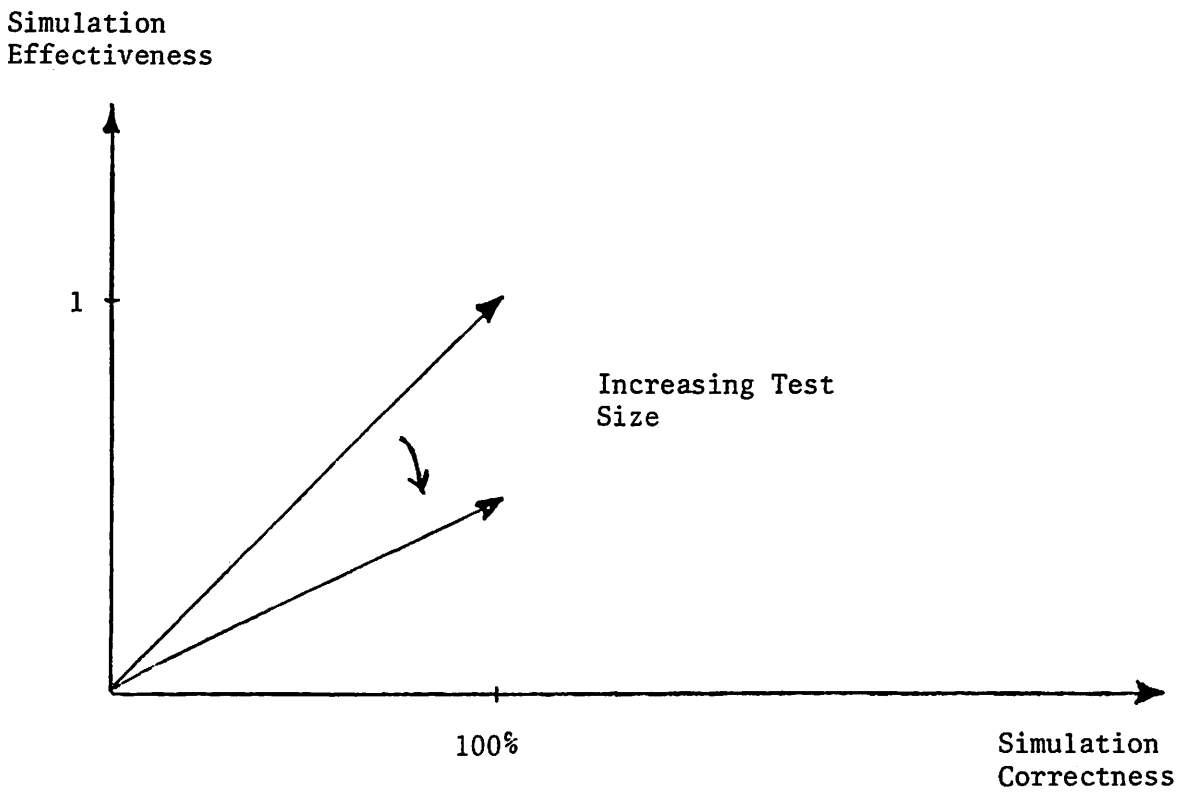


Figure 4-2: The Relationship Between Simulation Effectiveness and Simulation Correctness

First, errors fall into three general classes and twelve types within these classes. The types are:

1. LOGICAL CLASS OF ERRORS

   - Missing Event
   - Wrong Event
   - Wrong Order of Events
   - Missing Conditional Test
   - Wrong Conditions Tested

2. TIMING ERRORS

   - Simultaneous Events out of Phase
   - Missing Time-out Check
   - Event Late
   - Event Early

CONCURRENT ERRORS

   - Wrong Simultaneous Events
   - Resource Conflicts
   - Wrong Order of Events across Parallel Branches

Some errors were unable to be classified because not enough information was provided by the designers.

An example of a wrong event is reading from the wrong register. The wrong order of events (when no concurrency is present) is exemplified by using an old value from a register prior to writing in the correct value. Failure to check for overflow after an ADD is an example of a missing conditional test.

An example of a wrong conditions tested is testing for a positive value when the test should have included positive and zero values.

An example of a missing time-out check is the circumstance where an exception condition is signalled, and the responding hardware is so late the condition has changed or disappeared; the responding hardware should have timed out. A late event can occur, for example, when values arrive at register inputs too late to be clocked into the register correctly. A device which always responds later than other devices due to physical separation is an example of simultaneous events out of phase. Early events occur at interfaces between functional units; an example is a protocol signal arriving before data, instead of afterwards.

Wrong simultaneous events is a broad category of errors which includes producing an incorrect control flow by executing two events in parallel, both of which alter the control flow in different ways. Resource conflicts are exemplified by conditional tests on a flag set/ reset by two separate events for two different reasons. Wrong order of events can easily occur, for example, when "valid data" is signalled before data is actually valid. Deadlock occurs when events get out of order across parallel branches.

In addition, designers in all three major companies described PERFORMANCE errors as situations where desired performance was not met, usually due to unexpected concurrent events. This study does not consider these errors directly, since our focus is on a strict definition of correct behavior which does not include performance. However, it is an issue which should be explored further.

When simulation was performed prior to prototype testing, estimates of error coverage ranged from 10% to 30%[2] . Logical errors were most likely to be caught by simulation, and concurrent and timing errors about equally

---

[2]Some companies only reported "difficult" errors so this number actually indicates error coverage of the more troublesome design errors.

unlikely of being trapped this way. One company reported data-dependent logical errors as least-likely logical errors to be detected by simulation. Virtually only a handful of errors were caught by special-purpose hardware or simulation code designed just for that purpose.

Responses indicated that the same types of errors reoccurred infrequently in designs done by each responding designer but more than once. There were no obvious differences between these responses across error types.

Incorrect specifications resulted in errors as little as 10% and as much as 34% of the time. There seemed to be a general belief that most specification errors occurred when describing interaction across functional units.

The errors resulted in wrong data, wrong control flow, deadlock, system crashes and performance problems. Wrong control flow was the likely outcome of a given error, and bad performance the least likely. The errors occurred during normal processing, and during exception handling. Logical errors were most likely to occur during normal processing, and concurrent errors least likely to occur then, with timing errors falling somewhere in between.

A large number of errors were reported on as being due to missing conditional tests or branches. It is possible that this question was misunderstood to refer to simulation tests instead of tests on status while the hardware itself is executing.

Many of the errors reported on were timing dependent due to some type of asynchronous interaction, particularly I/O. In general, errors due to functional unit interaction seemed to dominate.

Companies reported that many of the errors were data-dependent; some errors were visible only when the data caused exception - handling mechanisms to be involved. Errors were introduced when previous problems were corrected and

errors may have been related to or masked by other errors, although the proportion of these varied widely across companies.

Two companies analyzed major causes of errors; they agreed that inadequate specifications, lack of integration/communication between departments, and the objective to perform tasks in parallel led to many of the design problems.

The importance of the survey results is that we have confirmed our intuitive beliefs about design errors. We can treat errors in sets according to class and type. We can focus on data-dependent logical errors, timing errors caused by asynchronous interaction and concurrent errors. Specification techniques require some attention, and exception handling hardware should get disproportionate attention. Functional-unit interaction requires careful analysis.

Now that this broad survey has been performed, a more specific, quantitative study of concurrent errors will be undertaken. The latter survey will provide guidelines for concentration on error types and heuristics for test-case selection.

5. SIMULATION EXAMPLES

To gain an understanding of the errors which commonly occur in hardware designs and how they show up during simulation, we carried out some example simulations. We wrote a description, using the hardware descriptive language SLIDE, of two devices communicating with each other, both with and without injected errors. The examples are included here to provide the reader with concrete instances of the error types described earlier, as well as to illustrate their manifestation via simulation outputs.

In specific, two devices, DEVA and DEVB communicate with each other over a 12-bit data bus "data", and two control lines, "for" and "bac". The communication protocol is as follows:

First DEVA places some data on the data bus. DEVB then reads the data on the bus, and, depending on the value of the two least significant bits, modifies it and places it back on the data bus. DEVA then reads the data and stores it. This process is then repeated. The configuration is shown in Figure 5-1 and the timing of the protocol is shown in Figure 5-2. The hardware descriptions and simulation runs are attached as Appendix B.

The hardware described above has been simulated using the SLIDE simulator and the values placed on the "data" bus. "for" and "bac" have been observed as a function of time. We then injected errors into the description, one at a time, and simulation was carried out again. The errors injected were:

- Timing Error: The data is read by DEVB from the "data" bus too early, even before DEVA places the data on the bus. (Event early). This error is created by changing a delay of 20 nanoseconds to 5 nanoseconds in the DEVB process.

- Logical Error: One of the statements in a conditional branch is omitted. This statement modifies the value of the least significant bit of the data read from the bus, before it is placed back on the "data" bus. (Missing Event). We create this error by omitting the statement which complements the data.

- Concurrent Error: A particular register is being written to and read at the same time. This is an example of a resource conflict. We create the write- read error by writing an internal register at the same time it is being read, instead of immediately before. Another concurrent error injected is a "write-write" kind of resource conflict i.e., a register being written to at the same time from two different places.

The only error caught by the simulator was the "write-write" resource conflict. However, all of the errors cause the output of the simulation runs to be modified. This leads us to the conclusion that such errors are detectable given proper simulation test inputs.

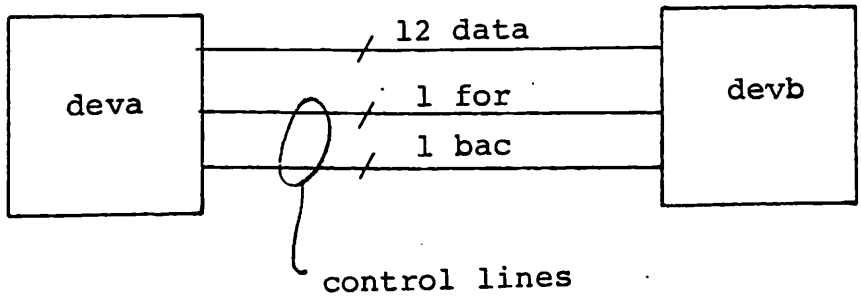


Figure 5-1: The Hardware Example Used for Simulations

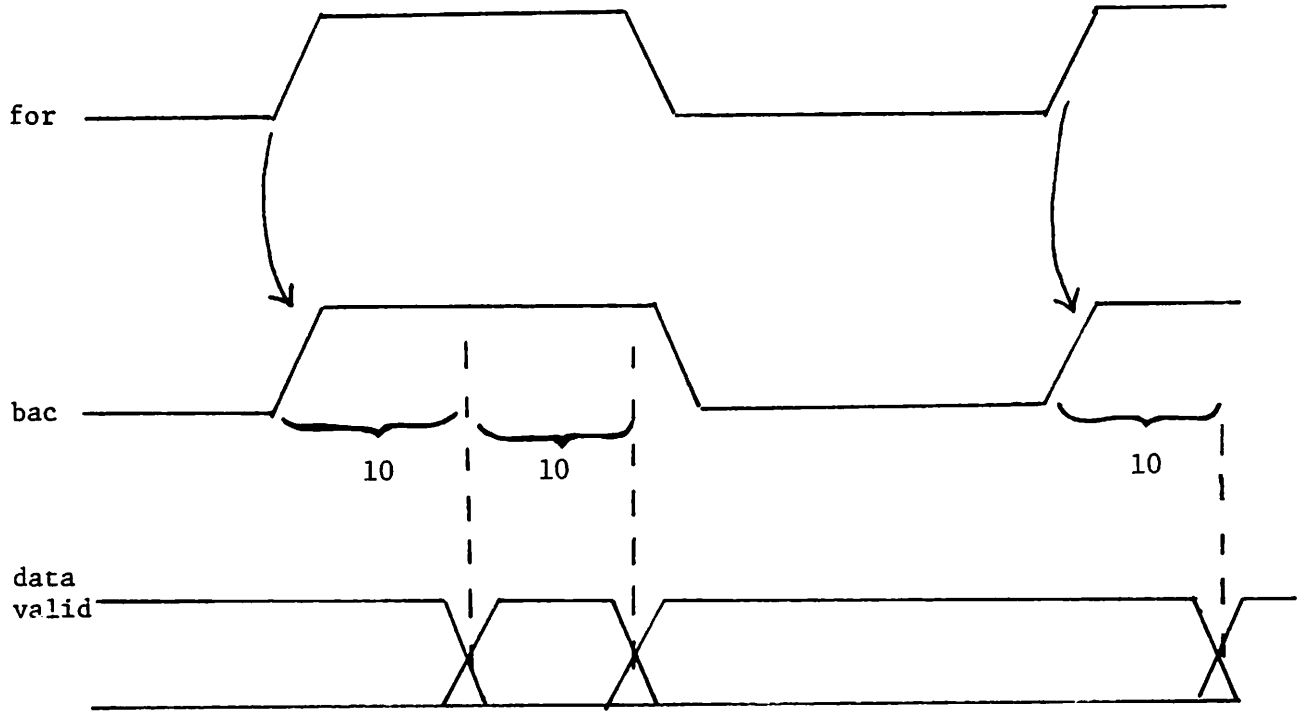


Figure 5-2: Timing of the Protocol for the Example Hardware

6. THE THEORY OF ERROR DETECTION

Once an error classification had been done, a more formal definition of each error type was required, in order to determine what kind of information must be extracted from a hardware description in order to guarantee error detection. In specific, proofs were undertaken to show what information was required to detect each type of error, and under what circumstances errors could be guaranteed to be detected.

The approach has been to create an arbitrary hardware description in a hardware descriptive language, and then to mutate the description by injecting an error of a certain type. Then, both the original and mutated descriptions have been processed to derive the abstract behavior. The proof technique shows exactly what information is required to detect the error in the abstract behavior, and under what circumstances errors cannot be detected, by attempting to prove that the two abstract behaviors are equivalent.

It is important to point out here that these proofs are not the direct basis for a practical technique for error detection; rather, they serve a number of indirect purposes:

- They give us experience in understanding the theory behind error detection, and how errors manifest themselves.
- They illustrate precisely under what conditions errors can be concealed.
- they force us to formalize error types in terms of a specific hardware descriptive language, and to indicate where in specific descriptions they would occur.
- They indicate precisely how each error type should be manifested in a symbolic simulation.

With this motivation, we now present an introduction to the specific hardware descriptive language we use, the Behavior Expressions we use to express abstract behavior, and the proofs themselves.

6.1. Introduction to ISPB

The hardware descriptive language used here, ISPB, is a basic form of ISP. It is block-structured, ALGOL-like, with ten possible action types. These actions are shown below, in Table 6.1.

Action	Definition
u <-- e	evaluate expression e and place in u.
x_i <-- e	evaluate expression e and place in x_i.
u <-- x_i	read x_i and place in u.
leave f_k	exit procedure f_k and return to calling point.
restart f_k	return to beginning of procedure f_k and continue execution.
Skip	Skip to the next action
if b then A_1 else A_2	If the Boolean expression b is true execute A_1, otherwise execute A_2.
A_1; A_2	The execution of actions A_1 and A_2 are order-independent.
A_1 next A_2	Complete action A_1 before beginning A_2.
Call f_k	Call procedure f_k.

Table 6-1: ISPB Actions

The following discussion in this and the next section is quoted directly from [4], the original thesis research by McFarland. We have taken the liberty of paraphrasing parts of the discussion, in order to be brief.

In ISPB all the shared variables are referred to as GLOBAL variables or "external" variables. The lower case x is reserved for these variables. The local or internal variables are referred to by u,v or w. [3] An ISPB program

---

[3] The distinction between global and local variables will be useful when concurrent errors are considered later in the research. At this point they are an aid in abstracting away unnecessary information.

consists of a series of procedure definitions. Each procedure definition is of the form ⟨identifier⟩: = ⟨action⟩. Executing the program means invoking (calling) the MAIN procedure. Invoking procedure f means executing the action A which is defined to be the body of f by the definition f:=A. The interaction with the environment is through the global variables. The local variables determine the state of the machine.

Definition 1:

An event θ in ISPB is a triplet of the form ⟨read, x_i, c⟩ or ⟨write, x_i, c⟩ or λ(the empty event) where x_i is a global variable and c is a constant value in the domain of x_i. An event is a fully interpreted interaction with the environment.

Definition 2:

A history η is a sequence of events. It may be finite or infinite.

Definition 3:

A behavior is a set of histories.

Definition 4:

If M is a machine description in ISPB, Bh(M) is the behavior of M, i.e., the set of histories generated by M.

6.2. Introduction to Behavior Expressions

According to McFarland, the behavior of a machine is its set of histories, each of which is a sequence of events [5]. An event is either a read or a write operation involving a global variable. This behavior is represented by a Behavior Expression (BE) which is actually a set of regular expressions, augmented by predicates to achieve partial ordering and conditional occurrence. Since a machine can have a large set of histories, some of which are repetitive, it is convenient to use regular expressions to represent its

behavior.

We use Behavior Expressions here for a different application than McFarland, illustrating their generality. We also propose extensions to BE's to cover timing and concurrency.

Predicates allow each "event" in an expression to be dependent on the past history of inputs to the program and the number of times any loops in which it is embedded have been "executed." The latter dependency is made possible by assigning a "loop counter," which is an integer variable different from any program variable, to each loop in the expression.

A predicate must not be confused with a condition/action statement. A predicate is the condition which holds after a particular event has occurred. It is not analogous to the execution of a program where an event occurs if the condition is true.

For ISPB, an event schema E is a string of the form λ, $R(x_i)$ or $W(x_i)$ for $x_i \in \underline{x}$. An atomic BE is a pair of the form E:P where E is an event schema and P is a predicate. For an atomic BE of the form E:P to be matched by an event θ it is certainly necessary for the type of event (read, write or null) and the name of the global variable to match the event schema E. Furthermore an event cannot by itself satisfy an atomic BE, but a history-event pair can.

We can extract the behavior from the description and represent it at the appropriate level of abstraction, in the form of Behavior Expressions.[4] There are two steps involved in forming a Behavior Expression which models an ISPB program. First atomic BE's must be formed which correspond to the atomic actions executed by the program. Then these must be composed into complex

---

[4]This process could be considered a formal version of symbolic simulation.

expressions to show sequencing, looping and so on. The second step is straightforward, since ISPB complex actions translate one-to-one into BE operators. An ISPB conditional statement translates into a "+"(OR) in the corresponding BE, a "next" to a "." (sequence), a ";" to a "||" (order independence) and a procedure call to a loop.

The atomic Behavior Expression for an atomic action must contain an event schema consistent with the type of action, a predicate which reflects the conditions which must hold when the action is executed, and, if a global variable is involved, the values read or written. The first step in finding the correct predicates involves computing pre- and postconditions for every action in the program. The precondition of an action is a logical formula which states what must be true of the internal variables, the external variables and the loop counters just before the action is executed. The postcondition is a similar formula describing the state of the system and its history just after the action has been executed.

Three functions T,L and R are used to find the pre-and post-conditions. The principal function T, takes an action A and a precondition P for that action and produces a postcondition for A. It is written T[A](P). If A is a composite action, the postcondition is computed from T applied to its component actions, using the appropriate preconditions for those components. If A is a procedure call of the form "call f" where f:=A', then the postcondition for A and the precondition for A', the body of f, depend on the preconditions holding at leaves and restarts[5] of f, which are imbedded in A'.

The purpose of the other two functions L and R is to compute those "leave"

---

[5] "Leave" translates control up an arbitrary number of nested levels out of the procedure named in the "leave" statement. "restart" transfers control up an arbitrary number of nested levels to restart the procedure named in the "restart" statement.

and "restart" conditions respectively. L[A,f](P) is a predicate which is the disjunct of all the preconditions of all the "leave f" actions which may be executed, when A is executed with a precondition of P. In other words L[A,f](P) is a predicate which is satisfied by any state which may be in effect when a "leave f" is executed within A, after A has been started in a state satisfying the precondition P. Similarly R[A,f](P) is a predicate characterizing all states which may hold when a "restart f" is executed within A, with A having been started in some state satisfying P. All the possible actions in ISPB and the definition of T and B (Behavior Expression) for them are listed below.

The Pre-to-Postcondition Transformer T

A	T[A](P)
$u \leftarrow e$	$\exists u'\ P\langle u \leftarrow u' \rangle \wedge u = e\langle u \leftarrow u' \rangle$
$x_i \leftarrow e$	P
$u \leftarrow x_i$	$\exists u', j'_i\ P\langle u \leftarrow u', j_i \leftarrow j'_i \rangle \wedge j_i = j'_i + 1 \wedge u = x_i(j_i)$
leave f_k	false
restart f_k	false
skip	P
if b then A_1 else A_2	$T[A_1](P \wedge b) \vee T[A_2](P \wedge \sim b)$
$A_1;A_2$	$\exists \underline{w}'_1, \underline{w}'_2 (T[A_1]((P \wedge \underline{w}_1 = \underline{w}'_1)\langle \underline{w}_2 \leftarrow \underline{w}'_2 \rangle) \wedge T[A_2]((P \wedge \underline{w}_2 = \underline{w}'_2)\langle \underline{w}_1 \leftarrow \underline{w}'_1 \rangle))$
A_1 next A_2	$T[A_2](T[A_1](P))$
call f_k	$\exists I \{L[A_k, f_k](P_k(P,I)) \vee T[A_k](P_k(P,I))\}$

where $f_k := A_k$

Here P_k is defined to be the unique least fixpoint of the equation

$P_k(P,I) = ((I = 1) \wedge P) \vee ((I > 1) \wedge R[A_k, f_k](P_k(P, I-1)))$

DEFINITION OF B

A	B[A](P)
$u \leftarrow e$	$\Lambda:/T[A](P)/$
$x_i \leftarrow e$	$W(x_i):/(P \wedge x_i = e)/$
$u \leftarrow x_i$	$R(x_i):/(T[A](P))/$
leave f_k	$\Lambda:/P/$
restart f_k	$\Lambda:/P/$
skip	$\Lambda:/P/$
if b then A_1 else A_2	$B[A_1](P \wedge b) + B[A_2](P \wedge \sim b)$
$A_1;A_2$	$B[A_1](P) \parallel B[A_2](P)$
A_1 next A_2	$B[A_1](P) \,.\, B[A_2](T[A_1](P))$
call f_k	$[B[A_k](P_k(P,I_k))]^* I_k$

where $f_k := A_k$

Table 6-2: Definitions of T and B, taken from [4]

P_k is defined to be the unique least fixpoint of the equation

$$P_k(P,I) = ((I=1 \wedge P) \vee ((I>1 \wedge R[A_k, f_k](P_k(P,I-1)))$$

If A is an atomic action $L[A,f_n]$ is false unless A is of the form "leave f_n". In any case $L[A,f_n](P) = P$. Similarly R is false for any atomic action except a restart. For complex actions R and L are composed from R and L applied to the component actions in much the same way as T.

For a procedure f_n with body A_n, the precondition of A_n is the precondition P to the call of f_n if it is the first time through the loop $(I_n = 1)$[6]. Otherwise it is the logical sum of all the restart conditions from the last pass through the loop (I_n-1), since when a "restart f_n" is executed with the machine in a certain state, the body of f_n is entered again with the machine still in that state. this gives rise to a recursive equation for P_n, the precondition to the body of f_n:

$$P_n\ (P,I_n) = (I_n = 1)\wedge P\vee((I_n>1\wedge R[A_n,f_n](P_n\ (P,I_n-1)))$$

The pre- and post- conditions which involve internal as well as external variables are converted to predicates over only external variables by "projecting out" the internal variables using existential quantification. This is called "closing" the predicate and is defined as follows:

Definition 5:

(Closure of a Predicate). Let $\underline{V}$ be the set of internal variables for an ISPB program and $\underline{X}$ the set of external variables. Let X and J be the sets of input-sequences and input sequence indexes generated from $\underline{X}$, and I the set of loop counters for the program. Let P be a predicate over $\underline{V}$, X, J and I. Then the closure of P is

$$/P/ = \exists\ \underline{V}\ P$$

---

[6]We have altered McFarland's notation here from lowercase L to I to distinguish it more readily from the numeral one

For example, if $P = x_1(j_1-1) = v \wedge 3*x_1(j_1) = u \wedge x_2 = u+v$ then

$$/P/ = \exists\, v,u\ P \subseteq (x_2 = x_1(j_1-1) + 3*x_1(j_1))$$

We now define logical and timing behavior:

Definition 6:

A machine M exhibits correct logical behavior if and only if Bh(M)<===> Bh(M') where M' is the specified machine and Bh(M') is the specified behavior. Bh(M) and Bh(M') can be represented by Behavior Expressions.

Definition 7:

Timing behavior of a machine is correct if and only if Bh(M)<===>Bh(M') and for each atomic event $\theta_{i,j}' \in Bh(M')$ which contains a time interval $[t_1', t_2']$ as part of the predicate there is a corresponding atomic event $\theta_{i,j} \in Bh(M)$ which contains a time interval $[t_1, t_2]$ where $t_1 \geq t_1'$ and $t_2 \leq t_2'$.

6.3. The Application of Behavior Expressions

We consider an arbitrary description written in ISPB, inject errors into it and show how these errors manifest themselves in the Behavior Expressions (BEs)for a particular description. In particular we shall consider an error in each of the types of actions of ISPB individually and show how these errors may alter the Behavior Expressions. We do formal proofs leading to this conclusion: given the behavior of an arbitrary description written in ISPB without an error, we can derive the Behavior Expression for the program with an injected error which is different in a predictable way from the first Behavior Expression.

LOGICAL ERRORS

We now apply Behavior Expressions to descriptions which contain logical errors.

Let us consider a common example of a logical error such as a missing

conditional test. Let us assume that if the condition is true then the corresponding action is to write a certain value to a global variable. If the condition is false then the action is to skip the write. Let us assume that this particular conditional test is missing. If we extract the behavior of the machine and represent it in the form of Behavior Expressions, the predicate corresponding to the event of writing to the variable will be incorrect, because of the missing conditional. Let us now consider the situation wherein the conditional branch is missing but the conditional test itself is present. Referring to the above example the missing conditional branch is the action of writing to the variable, which will be missing from the Behavior Expression, because the Behavior Expression of the machine covers the entire set of histories of that machine. This information can be fed back to the designer who can then decide whether the design meets its specifications or not, thereby detecting errors. Now it should be apparent that if the designer specifies what information should be abstracted into a Behavior Expression, the behavior at the appropriate level of abstraction can be derived easily.

We now consider specific examples of errors:

1. missing conditional test and
2. missing event (missing conditional branch)

We will write the Behavior Expression for a small program and see what effect the errors could have on the expression. As we described above, a missing conditional branch would lead to the absence of a regular expression from the Behavior Expression, whereas a missing condition would lead to an incorrect predicate. Recall that each atomic Behavior Expression is a pair E:P corresponding to "event: predicate".

Let us now consider the following program, which sets a global flag x_{start} to one, and then depending upon the value of a variable x_{in} writes out a value

to x_{out}.

f_{write} := v <— x_{in}
if v = 0 then x_{out} <— 1 else skip

main := x_{start} <— 1 next
call f_{write} next
x_{start} <— 0

The Behavior Expression for the above program is as follows:

$[W(x_{start}):(x_{start}=1 \wedge j_{in}=0)]$
$.[R(x_{in}):(x_{start}=1 \wedge j_{in}=1)]$
$.[W(x_{out}):(x_{start}=1 \wedge j_{in}=1$
$\wedge x_{out}=1 \wedge x_{in}(j_{in})=0)]$
$+ [\lambda:(x_{start}=1 \wedge j_{in}=1 \wedge x_{in}(j_{in})\neq 0)]$
$.[W(x_{start}):(x_{start}=0 \wedge j_{in}=1 \wedge (x_{out}=1 \vee x_{in}(j_{in})\neq 0))]$

"." indicates a sequence as in a regular expression.

This program could be a part of a machine description and the global variable x_{in} could have been read a number of times before. Therefore, $x_{in}(j_{in})$ corresponds to the last value of x_{in} read.

Now let us see what happens if the condition v=0 is missing but the action of writing to the variable x_{out} is still present. In this case the Behavior Expression will be modified as follows:

$[W(x_{start}):(x_{start}=1 \wedge j_{in}=0)]$
$.[R(x_{in}):(x_{start}=1 \wedge j_{in}=1)]$
$.[W(x_{out}):(x_{start}=1 \wedge j_{in}=1$

$\wedge\ x_{out}=1\ )]$

$.[W(x_{start}):(x_{start}=0 \wedge j_{in}=1 \wedge x_{out}=1)]$

Thus the missing condition results in an incorrect predicate and incorrect actions. Let us see the result if the conditional branch of writing to the variable x_{out} is missing. In this case the Behavior Expression appears as shown below

$[W(x_{start}):(x_{start}=1 \wedge j_{in}=0)]$

$.[R(x_{in}):(x_{start}=1 \wedge j_{in}=1)]$

$.[W(x_{start}):(x_{start}=0 \wedge j_{in}=1)]$

In this case the expression corresponding to the action of writing to the variable is missing.

TIMING ERRORS

The same procedure that was used for logical errors, with one modification, can be applied to timing errors by introducing time dependency in the predicate. The timing information can be derived from the control flow information or from the hardware description directly. A method for specifying timing dependencies can be found in [1].

Let us see how Behavior Expressions can be used to detect timing errors. We will consider the same example as above. The theory of Behavior Expressions is independent of the particular details of the interpretation of the data types of the hardware descriptive language. Thus x_{in} in the above example could very well have been a register. Let us assume that the timing analysis is performed for the above program and we reach the conclusion that the value from x_{in} must be read after a certain time T. The time T may depend

upon the propagation delay through the register and some other parameters. The Behavior Expression considered above could be modified by introducing this time dependency in the predicate

$[W(x_{start}):(x_{start}=1 \wedge j_{in}=0)]$

$.[R(x_{in}):(x_{start}=1 \wedge t_{x_{in}} > T \wedge j_{in}=1)]$

$.[W(x_{out}):(x_{start}=1 \wedge j_{in}=1$

$\wedge x_{out}=1 \wedge x_{in}(j_{in})=0)]$

$+ [\lambda:(x_{start}=1 \wedge j_{in}=1 \wedge x_{in}(j_{in})\neq 0)]$

$.[W(x_{start}):(x_{start}=0 \wedge j_{in}=1 \wedge (x_{out}=1 \wedge x_{in}(j_{in})\neq 0))]$

Again the derived behavior could be studied and decisions could be made whether it is correct.

We present an example proof here. The remaining proofs can be found in Appendix D.

Assignment to an Internal Variable

u <— e

The above atomic action assigns the value of expression e to the local variable u. The errors which are possible are that the atomic action could be missing, the value assigned to u could be incorrect, the variable to which the assignment occurs is wrong or the position in which the assignment occurs is wrong. The last possibility shall be considered separately since it forms a different and general class of errors, i.e., wrong sequencing of events.

Consider the Behavior Expression of an arbitrary program which has u<— e as one of its statements. (The BE of u <— e is $\lambda:/T[u<— e](P)/$, where λ is the empty event):

$$B^1.\lambda/T[u\langle\!-\!-\ e](P)/.B^2.B^3 \qquad (1)$$

where B^1 and B^3 are some complex and arbitrary Behavior Expressions and B^2 is a Behavior Expression corresponding to any of the atomic or complex actions listed in the previous section.

We have assumed that the program which is represented by the expression listed in 1 has the assignment as an intermediate statement. The other possibilities are that the assignment could be either the first or last statement of the program, in which case the Behavior Expression would be either

$$\lambda:/T[u \langle\!-\!-\ e](P)/.B^1.B^2 \qquad (2)$$

or

$$B^1.\lambda:/T[u \langle\!-\!-\ e](P)/ \qquad (3)$$

The expression in 3 is equivalent to B^1 using λ-reduction [4] and thus any error in the assignment is not going to affect the Behavior Expression. This means that if an assignment to a local variable is the last action in a hardware description, it does not affect the externally visible behavior. This is as expected, since local variables are only observed when they affect global variables.

Thus we have to consider expressions in 1 and 2 only. The proofs for both of them are the same and thus we shall consider the expression in 1. We consider the following case:

Missing atomic action followed by $x_i \langle\!-\!-\ e$

We shall consider what happens to the behavior expression if the assignment $u \langle\!-\!-\ e$ is missing. We will have to consider that the assignment could be followed by any of the atomic or complex actions. Thus we shall consider each

case separately.

Case (a): Assignment followed by x_i <— e. In this case the Behavior Expression would be

$$B^1.\lambda:/T[u \text{ <— } e](P_1)/.\ \ W(X_i):/P_2 \wedge x_i = e/.B^3 \tag{4}$$

/T[u <— e](P$_1$)/ is the postcondition of the assignment u <— e and P_1 is the precondition of the assignment. P_2 is the precondition of the statement x_i<— e.

Now P_2 = /T[u<— e](P$_1$)/; i.e., the postcondition of the assignment u <— e is equal to the precondition of the assignment x_i <— e.

Now if the assignment is missing, P_2 is going to be incorrect, unless $u' \equiv u$, where u' = value of the variable after assignment and u the value of the variable before the assignment. This special case is obiously the only one where a missing assignment to a local variable cannot be detected by a change in the BE.

Thus P_2 will be changed to P_2'. Hence the Behavior Expression will be

$$B^1.\ \ W(x_i):/P_2' \wedge x_i = e/.\ \ B^3 \tag{5}$$

instead of

$$B^1.\ \ W(x_i):/P_2 \wedge x_i = e/.\ \ B^3 \tag{6}$$

where P_2 = /T[u <— e](P$_1$)/ and P_2' = /P$_1$/.

The expression in 4 reduces to the expression in 6 by λ-reduction. Hence the missing atomic action is going to manifest itself in the predicate of the write action.

7. RESEARCH DISCUSSION AND FUTURE DIRECTIONS

We now address three continuing research topics:

1. Potential Timing and Logical Error Coverage
2. A Proposed Methodology for Logical Error Detection
3. Extensions of Behavior Expressions for Concurrent Behavior

Timing and Logical Error Coverage

In order to approach 100% coverage of logical and timing errors, hardware descriptions must be processed either using exhaustive simulation or symbolically so that all behavior possibilities can somehow be represented to the designer. Logical error coverage using symbolic simulation (for the classes of errors and types of errors in these classes covered in this report) can approach 100% if we make one assumption. This assumption is that errors occur singly, do not interact with each other, and do not mask other errors. (Based on our survey results, this is a legitimate assumption). Even so, however, the proofs which guarantee detection of logical errors under most circumstances are a research tool. Applying such formal analysis techniques in practice may require limiting assumptions which restrict detection to less than 100% or which restrict error types which can be detected. The proofs in Appendix D do tell us what information must be extracted from a hardware description in order to detect logical errors, and give us some intuition as to the difficulties of detection of each type error.

Timing errors within a process or module can be guaranteed to be detected by formal processing methods (e.g. [2, 3]) as long as no asynchronous interaction is present. Time variables could be included in the predicates of the Behavior Expressions in order to prove this, as logical error detection was proven. However, timing errors across modules (or across concurrent control flows) have not been addressed thus far in the research and form an important component of the next stage.

A Proposed Method for Logical Error Detection

Logical error detection is successful using current simulation methods. However, coverage could approach 100% if more formal techniques for processing the hardware description were used. A proposed approach is to mechanically process the description to extract its abstract behavior (a type of symbolic simulation). This information could be fed back to the designer who could then decide if that is exactly what he meant while writing the description. A problem with this approach is that the Behavior Expression would be large and complex in an actual design. Alternatively a lot of time and effort could be saved if the designer could specify the expression for correct behavior, in which case the behavior at the appropriate level of abstraction could be extracted. Using this method a lot of redundant information could be avoided, and the abstraction process would be simpler. A hybrid method could also be used to symbolically simulate the description, interactively asking the designer questions about the correct (intended) behavior, as the simulation progressed.

This process may involve conversion of non-procedural descriptions to procedural descriptions. Some thought has gone into this process.

Concurrent Behavior Expressions

Concurrent Behavior Expressions should be capable of expressing the following situations:

- Wrong combination of simultaneous events
- Resource conflicts
- Wrong ordering of events

The partial ordering and resource conflicts are both handled by predicates. In either case the predicates may not be exclusive. Wherever two or more branches may be executed in parallel the predicates in one branch may be

dependent on the predicates in some other branch, producing a required partial ordering. The predicates used to express resource conflicts could be Boolean expressions.

Let us review some of the problems introduced by concurrent errors. Consider the following situation where there are two parallel branches, and within each of these branches there are certain sequences of events. These events in the parallel branches could be partially ordered with respect to each other. Consider the specific situation where there are three events in each branch, as shown in Figure 7-1 below.

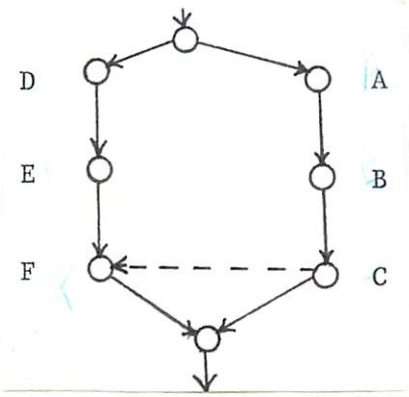


Figure 7-1: Potential Parallelism of Events

A, B, and C can occur in parallel with D, E, and F but there is one restriction placed on the occurrence of F. F can occur in parallel with C, but only after C has started. In other words F cannot occur before C. Thus the predicate attached to event F will be dependent on C. The requirement for the predicate is that it should be able to express the situation wherein C has started to occur. We know of operators which can represent parallel or sequential occurrence but not the situation which specifies that an event has started to occur.

A second problem is how to specify situations which allow/disallow resource conflicts. Our approach is to consider individual actions such as reads or writes of global variables as using resources. Thus we attach to such events predicates which disallow any resource conflicts. We also assume over the resources a certain kind of locality so that we handle resource conflicts only

in the parallel branches under consideration. We do not concern ourselves with other parent parallel branches of the immediate parallel branches. The resource conflicts in the parent branches would have been taken care of by their predicates.

The wrong combination of simultaneous events is more difficult to express. Basically, errors occur here when two simultaneous (or parallel) events attempt to alter the overall control flow in different ways. A possible way of expressing correct behavior is to put specification of event occurrences into the predicates, to keep conflicting events from occurring simultaneously, or across two parallel branches. Otherwise, it is difficult to describe mutual exclusion without introducing artificial variables or a partial ordering to the events.

8. CONCLUSIONS

The conclusions to be drawn at this point in the research have all at least been alluded to in the previous sections. However, for completeness, we enumerate them here.

First, we produced a clear statement of the research problem. Then, we precisely defined what we mean by correct behavior. Next, we showed that design errors could be classified into types, and the survey results confirmed this. We showed with simulation examples how errors could be detected with proper attention to test inputs and simulation results.

A useful result of the survey was the fact that few of the errors interacted, or were masked by other errors. Thus, we could parallel the "single fault" assumption of testing hardware with a "single error" assumption. This further validates the proof results, since there we did not consider cases where errors interacted.

A major result was the proof that logical errors could be guaranteed to be

detected if precisely the right information was extracted from the hardware description in a formal manner. This theoretical result allows us to understand better what information must be derived from a hardware description in practice in order to provide acceptable levels of detection of each type of logical error.

The literature search confirmed our suspicions that little previous research was directly applicable, but that there were analogies in the software domain which could be exploited in the future research.

Finally, we have concluded that concurrent errors cause the majority of detection problems and hence should be focused on in future research.

9. ACKNOWLEDGMENTS

The author is indebted to Hillel Ofek and Ralph Bahnsen for their continued technical interaction during the course of this research. Vittal Kini has provided consulting on a regular basis, and has been helpful in all aspects of the research. Mike McFarland provided key ideas, and his consulting efforts have been extremely valuable to the research. All the survey respondents are gratefully acknowledged.

[1] Hafer, L., Parker, A.
A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic.
In Design Automation Conference Proceedings no. 18, pages 846-853. ACM SIGDA IEEE Computer Society-DATC, June, 1981.

[2] Hitchcock, R. B., et. al.
Timing Analysis of Computer Hardware.
IBM Journal of Research and Development 26(1):100-105, Jan., 1982.

[3] Hitchcock, R., Sr.
Timing Verification and the Timing Analysis Program.
In Proceedings of the 19th Design Automation Conference, pages 594-604. ACM SIGDA and IEEE Computer Society TC-DA, June, 1982.

[4] McFarland, M.
On Proving the Correctness of Optimizing Transformations in a Digital Design Automation System.
In Design Automation Conference Proceedings no. 18, pages 90-97. ACM SIGDA, IEEE Computer Society-DATC, June, 1981.

[5] McFarland, M.
Mathematical Models for Verification in a Design Automation System.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, July, 1981.

APPENDIX A

Design Error Survey

USC DEPARTMENT OF ELECTRICAL ENGINEERING-SYSTEMS

Professor Alice C. Parker
Dr. Vittal Kini
Vishal Wanchoo

15 April 1982

The purpose of this survey is to determine what sorts of design errors are made by hardware designers which are difficult to detect by conventional simulation methods. This information will be used to research new methods for deriving test data for simulation. In doing the research, the emphasis will be on the register-transfer and gate levels of design, and errors in arithmetic hardware will not be considered.

Because designers must prepare design inputs for simulation by writing (or flowcharting) the designs, we are interested not only in design errors per se but also in the errors made while writing descriptions. We are not interested in errors which can be detected while compiling or preprocessing the descriptions or even in a single simulation pass, but rather we are interested in errors which are detected at a later stage such as prototype testing. We would therefore like more information about errors which are timing dependent, errors which are detected only because of specific data inputs, and errors which occur due to the concurrent nature of the hardware.

Please answer questions 1 through 9 below for each error, assuming independent errors.

1. Describe the error.

2. Did the error occur because

   a. The architectural specification was incomplete, inadequate or misunderstood in the process of design or

   b. The design itself was wrong or

   c. Of an incorrect description of the design (ie. the design was understood correctly but described wrongly)?

3. Was the detection of the error data-dependent? By this we mean one of the following events occured:

   a. There were conditional tests on this data which caused a path in error to be executed only under certain conditions.

   b. The hardware error only occured when processing data which caused exceptions to occur,activating exception handling hardware which contained the design error. (Errors are a class of exceptions).

4. Was the error dependent on timing (ie) did it occur because of asynchronous interaction?

5. Has this type of error occured frequently in the designs you have done? Once,infrequently but more than once, or in many of the designs you do?

6. What was the severity of the error? Did it cause wrong data results, wrong control sequencing, system deadlocks, system crashes?

7. Was the error not trapped by simulation because it caused a side-effect that was not tested?

8. Did the error occur because there were operations or actions which interacted unexpectedly? Did these interactions occur as a result of normal processor execution or did they occur when the processor was handling exceptions?

9. Did the error occur because a conditional test was omitted, or did it occur because the conditional test was performed but one of the conditional branches was omitted?

10. Was the error caught by simulation? If not when was it caught and why didn't simulation catch it?

11. Was the error introduced by attemting to correct a previous problem?

12. Were the errors caught by special purpose hardware or simulation code designed just for that purpose?

13. Was there any relationship between errors? Did any of the errors mask other errors?

14. Other comments:

15. Were the questions in this survey clear to you?

APPENDIX B

Simulation Examples

```
! The following two SLIDE descriptions describe two devices communicating
  with each other over a data bus. The communication is controlled by
  a pair of control lines. !


MAIN PROCESS devb;                          ! First device

CLOCK 1;

LINE
   data<11:0>,                              ! data bus
   for<>,                                   ! control line
   bac<>;                                   ! control line

INIT B:0 WHEN bac EQL /;                    ! PROCESS B starts executing on
                                            ! the rising edge of bac.

PROCESS B;

REGISTER
     b<11:0>,                               ! 12 bit register used by process B.
     decs<1:0>;                             ! 2 bit register.

COMB
   zero<1:0> := '00,                        ! binary value 00 is assigned to zero.
   one<1:0>  := '01,
   two<1:0>  := '10;

BEGIN
  delay 20 NEXT                             ! delay by 20 clock cycles.
  b _ data NEXT                             ! contents of 'data' are transferred
                                            ! to 'b'
  decs<1:0> _ b<1:0>;                       ! two LSB's of 'b' are assigned  to
                                            ! 'decs'
  IF decs EQL zero THEN                     ! if 'decs' is zero then write 'b'
                                            ! onto 'data'
   BEGIN
    data_b NEXT
    for _ ≠ NEXT                            ! 'for' makes a transition from '1'
                                            ! to '0'
    bac _ ≠
   END
  ELSE IF decs EQL one THEN                 ! if 'decs' is one and
   BEGIN
    IF PARE(b<11:0>) THEN                   ! if parity of 'b' is even
     BEGIN
      b<0> _ NOT b<0> NEXT                  ! LSB of 'b' is complemented
      data _ b NEXT
      for _ ≠ NEXT
      bac _ ≠
     END
```

```
    ELSE                                    ! if 'decs' is one and
     BEGIN                                  ! if parity of 'b' is odd
       b<1> _ NOT b<1> NEXT                 ! LSB of 'b' is complemented
       data _ b NEXT
       for _ ≠ NEXT
       bac _ ≠
     END
   END
  ELSE IF decs EQL two THEN                 ! if 'decs' is two
   BEGIN
    data<11:6> _ b<5:0>;                    ! write 'b' onto 'data'
    data<5:0> _ b<11:6> NEXT                ! swapping bytes
    for _ ≠ NEXT
    bac _ ≠
   END
  ELSE                                      ! if 'decs' is three
   BEGIN
    data _ 0 NEXT                           ! all data lines are made '0'
    for _ ≠ NEXT
    bac _ ≠
   END
 END;

 BEGIN
  for _ / ;                                 ! 'for' transitions from '0' to '1'
  DELAY WHILE 1                             ! delay forever
 END
```

```
MAIN PROCESS devA;                          ! Second device

CLOCK 1;

LINE
  data<11:0>,                               ! Same external lines
  for<>,
  bac<>;

EAT REGISTER
  flag<>;

INIT Dummy:0 WHEN 1;                        ! Dummy process is always executing
                                            ! since its initiation condition is
                                            ! always true.
```

```
PROCESS Dummy;

        REGISTER
          soa<11:0>,
          a<11:0>;

        INIT Assign:0 WHEN flag EQL /;   ! Starting conditions for 'Assign',
        INIT Adata:0 WHEN for EQL /;     ! 'Adata', and 'Aread'
        INIT Aread:0 WHEN bac EQL ≠;

PROCESS Adata;
BEGIN
  bac _ /;
  delay 10 NEXT                          ! delay by 10 clock cycles
  data _ soa                             ! write onto data lines
END;

PROCESS Aread;

BEGIN
  a _  data;                             ! read from data lines
  DELAY 75 NEXT                          ! and delay 75 clock cycles
  for _ /
END;

PROCESS Assign;

BEGIN
  soa _ #7775 NEXT                       ! 'soa' is assigned octal value 7775
  flag _ ≠                               ! lower flag
END;

BEGIN !dummy process!
  DELAY WHILE 1                          ! idle forever!
END;

BEGIN !main process!
  flag _ /;                              ! raise flag to start 'Assign'
  DELAY WHILE 1                          ! idle forever!
END
```

```
! This description of Devb contains an error. This is an example of a
  timing error. The value of "data" is being read too early. The contents
  of "soa" are supposed to be written into "data", after which the value
  of "data" is read and written into register "b". In the description
  shown below the value of "data" is being read and transfered to "b"
  even before "data" is written to from "soa". This error is not caught
  by the simulator but the output of the simulation run is different from
  the expected output !

MAIN PROCESS devb;

CLOCK 1;

LINE
   data<11:0>,
   for<>,
   bac<>;

INIT B:0 WHEN bac EQL /;

PROCESS B;

REGISTER
     b<11:0>,
     decs<1:0>;

COMB
   zero<1:0> := '00,
   one<1:0>  := '01,
   two<1:0>  := '10;

BEGIN
  delay 5 NEXT                    ! delay has been changed from 20 to 5
                                  ! thus causing data to be read early.
  b _ data NEXT
  decs<1:0> _ b<1:0>;
  IF decs EQL zero THEN
   BEGIN
    data_b NEXT
    for _ ≠ NEXT
    bac _ ≠
   END
  ELSE IF decs EQL one THEN
   BEGIN
    IF PARE(b<11:0>) THEN
     BEGIN
      b<0> _ NOT b<0> NEXT
      data _ b NEXT
      for _ ≠ NEXT
      bac _ ≠
     END
```

```
   ELSE
    BEGIN
      b<1> _ NOT b<1> NEXT
      data _ b NEXT
      for _ ≠ NEXT
      bac _ ≠
    END
  END
 ELSE IF decs EQL two THEN
  BEGIN
   data<11:6> _ b<5:0>;
   data<5:0> _ b<11:6> NEXT
   for _ ≠ NEXT
   bac _ ≠
  END
 ELSE
  BEGIN
   data _ 0 NEXT
   for _ ≠ NEXT
   bac _ ≠
  END
END;

BEGIN
 for _ / ;
 DELAY WHILE 1
END
```

```
! The description shown below contains an example of a logical error,
  a missing conditional branch.  One of the statements ( b<0> _ NOT b<0> )
  in a particular conditional branch( if decs EQL one ) is missing.
  This error is also not detected by the simulator but the output of the
  simulation run is different from the expected output. !

MAIN PROCESS devb;

CLOCK 1;

LINE
   data<11:0>,
   for<>,
   bac<>;

INIT B:0 WHEN bac EQL /;

PROCESS B;

REGISTER
     b<11:0>,
     decs<1:0>;

COMB
   zero<1:0> := '00,
   one<1:0>  := '01,
   two<1:0>  := '10;

BEGIN
  delay 20 NEXT
  b _ data NEXT
  decs<1:0> _ b<1:0>;
  IF decs EQL zero THEN
   BEGIN
    data_b NEXT
    for _ ≠ NEXT
    bac _ ≠
   END
  ELSE IF decs EQL one THEN
   BEGIN
    IF PARE(b<11:0>) THEN
     BEGIN
                                              ! b<0> _ NOT b<0> is missing
      data _ b NEXT
      for _ ≠ NEXT
      bac _ ≠
     END
```

```
    ELSE
     BEGIN
       b<1> _ NOT b<1> NEXT
       data _ b NEXT
       for _ ≠ NEXT
       bac _ ≠
     END
   END
  ELSE IF decs EQL two THEN
   BEGIN
    data<11:6> _ b<5:0>;
    data<5:0> _ b<11:6> NEXT
    for _ ≠ NEXT
    bac _ ≠
   END
  ELSE
   BEGIN
    data _ 0 NEXT
    for _ ≠ NEXT
    bac _ ≠
   END
 END;

 BEGIN
  for _ / ;
  DELAY WHILE 1
 END
```

```
! The description below contains an example of a concurrent error, a
  resource conflict. "b" is being written into and read at the same time.
  This is a 'read/write' type of resource conflict. This error is not detected
  by the simulator or by the simulation run. There is another error, however,
  a 'write/write' type of resource conflict which is caught by the simulator. !

MAIN PROCESS devb;

CLOCK 1;

LINE
   data<11:0>,
   for<>,
   bac<>;

INIT B:0 WHEN bac EQL /;

PROCESS B;

REGISTER
     b<11:0>,
     decs<1:0>;
```

```
COMB
   zero<1:0> := '00,
   one<1:0>  := '01,
   two<1:0>  := '10;

BEGIN
  delay 20 NEXT
  b _ data ;                          ! a NEXT statement after 'b _ data'
  decs<1:0> _ b<1:0>;                 ! is missing causing the next statement
  IF decs EQL zero THEN               ! to be executed in parallel with it.
   BEGIN                              ! Thus 'b' is being written to and read
    data_b NEXT                       ! in parallel.
    for _ # NEXT
    bac _ #
   END
  ELSE IF decs EQL one THEN
   BEGIN
    IF PARE(b<11:0>) THEN
     BEGIN
      b<0> _ NOT b<0> NEXT
      data _ b NEXT
      for _ # NEXT
      bac _ #
     END
    ELSE
     BEGIN
       b<1> _ NOT b<1> NEXT
       data _ b NEXT
       for _ # NEXT
       bac _ #
     END
   END
  ELSE IF decs EQL two THEN
   BEGIN
    data<11:6> _ b<5:0>;
    data<5:0> _ b<11:6> NEXT
    for _ # NEXT
    bac _ #
   END
  ELSE
   BEGIN
    data _ 0 NEXT
    for _ # NEXT
    bac _ #
   END
 END;

 BEGIN
  for _ / ;
  DELAY WHILE 1
 END
```

```
[PHOTO:  Recording initiated  Sat 29-May-82 3:10PM]

[Link from SPEAR, TTY 167]

 TOPS-20 Command processor 4(560)

! THE SIMULATION RUN BELOW IS OF THE EXAMPLE WITHOUT AN INJECTED ERROR

@r corr
SLIDE/Multi-Level Simulator Version 1.0
Welcome and Good Luck!!

#GET TRAP.IL
#ALL
VISHAL: DEVA    DATA FOR BAC FLAG;                !Vishal is an instance of Deva
VITTAL: DEVB    DATA FOR BAC;                     !Vittal is an instance of Devb
#SIMU                    !data is connected to data, for to for, bac to bac
%Simulation time parameters for VISHAL : DEVA may be bound now
%finished
%Simulation time parameters for VITTAL : DEVB may be bound now
%finished
#PR DATA                                          ! Probe data, for, bac, flag
#PR FOR
#PR BAC
#PR FLAG
#GO  J1
--->       0.000us FLAG                           ! Information about FLAG
 LOGIC=    6 SIZE=   0 PERIOD=       0.000us     ! SIZE= 0 means '1' bit
VALUES ON WIRE-    1                              ! value of FLAG is '1'
--->       0.000us FOR
 LOGIC=    6 SIZE=   0 PERIOD=       0.000us
VALUES ON WIRE-    1
--->       0.000us BAC
 LOGIC=    6 SIZE=   0 PERIOD=       0.000us
VALUES ON WIRE-    1
--->       0.001us FLAG                           ! refers to FLAG at .001us.
 LOGIC=    6 SIZE=   0 PERIOD=       0.000us
VALUES ON WIRE-    0                              ! value of FLAG is '0'
--->       0.011us DATA                           ! refers to DATA at .001us
 LOGIC=    6 SIZE=  11 PERIOD=       0.000us     ! size=11 means 12 bits
VALUES ON WIRE-    111111111101                   ! value is 111111111101
--->       0.023us DATA
 LOGIC=    6 SIZE=  11 PERIOD=       0.000us
VALUES ON WIRE-    111111111100
--->       0.024us FOR
 LOGIC=    6 SIZE=   0 PERIOD=       0.000us
VALUES ON WIRE-    0
```

```
—>        0.025us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.101us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.101us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.112us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.124us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.125us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.126us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.202us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.202us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.213us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.225us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.226us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.227us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.303us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.303us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.314us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.326us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
```

```
—>        0.327us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.328us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.404us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.404us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.415us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.427us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.428us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.429us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.505us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.505us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.516us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.528us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.529us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.530us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.606us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.606us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.617us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
```

```
—>        0.629us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.630us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.631us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.707us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.707us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.718us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.730us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.731us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.732us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.808us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.808us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.819us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.831us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.832us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.833us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.909us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.909us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
```

```
—>        0.920us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.932us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.933us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.934us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        1.000us
#EXIT

5 garbage collection(s) in 183 ms

End of SIMULA program execution.
CPU time: 8.30  Elapsed time: 57.06
@pop

[PHOTO:  Recording terminated  Sat 29-May-82 3:12PM]
```

```
[PHOTO:  Recording initiated  Sat 29-May-82 3:40PM]

[Link from SPEAR, TTY 167]

 TOPS-20 Command processor 4(560)

! THE SIMULATION RUN BELOW IS WITH AN INJECTED TIMING ERROR


@r timing
SLIDE/Multi-Level Simulator Version 1.0
Welcome and Good Luck!!

#GET TRAP J.IL
#BAD COMMAND.
#BAD COMMAND.
#ALL
VISHAL: DEVA     DATA FOR BAC FLAG;
VITTAL: DEVB     DATA FOR BAC;
#SIMU
%Simulation time parameters for VISHAL : DEVA may be bound now
%finished
%Simulation time parameters for VITTAL : DEVB may be bound now
%finished
#PR DATA
#PR  JFOR
#PR BAC
#PR FLAG
#GO  J1
```

```
—>        0.000us FLAG
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.000us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.000us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.001us FLAG
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.007us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    00000000000
—>        0.008us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.009us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.011us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
—>        0.085us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.085us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.093us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    11111111100
—>        0.094us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.095us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.096us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
—>        0.171us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.171us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.179us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    11111111100
—>        0.180us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.181us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
```

```
! The sequence of writes to
! the registers and lines is
! different from the expected
! output. The times at which
! the writes occur is also
! incorrect. The first write
! to DATA is 000000000000
! instead of 111111111101.
! All of this shows up in the
! simulation run but is not
! caught by the simulator.
```

```
—>        0.182us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.257us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.257us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.265us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.266us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.267us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.268us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.343us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.343us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.351us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.352us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.353us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.354us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.429us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.429us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.437us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.438us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.439us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—    0
```

```
—>        0.440us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
—>        0.515us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.515us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.523us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111100
—>        0.524us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.525us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.526us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
—>        0.601us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.601us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.609us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111100
—>        0.610us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.611us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.612us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
—>        0.687us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.687us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.695us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111100
—>        0.696us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.697us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
```

```
—>        0.698us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.773us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.773us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.781us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.782us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.783us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.784us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.859us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.859us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.867us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.868us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.869us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.870us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.945us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.945us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.953us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.954us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.955us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
```

```
-->        0.956us DATA
 LOGIC=    0 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    111111111101
-->        1.000us
#EXIT

0 garbage collection(s) in 235 ms

End of SIMULA program execution.
CPU time: 8.90  Elapsed time: 1:15.03
@pop

[PHOTO:  Recording terminated  Sat 29-May-82 3:42PM]
```

```
[PHOTO:  Recording initiated  Sat 29-May-82 5:22PM]

[Link from SPEAR, TTY 167]

 TOPS-20 Command processor 4(560)


! INJECTED ERROR IS A LOGICAL ERROR


@r logicl
SLIDE/Multi-Level Simulator Version 1.0
Welcome and Good Luck!!

#GET TRAP.IL
#BAD COMMAND.
#BAD COMMAND.
#ALL
VISHAL: DEVA     DATA FOR BAC FLAG;
VITTAL: DEVB     DATA FOR BAC;
#SIMU
%Simulation time parameters for VISHAL : DEVA may be bound now
%finished
%Simulation time parameters for VITTAL : DEVB may be bound now
%finished
#PR DATA
#PR FOR
#PR BAC
#PR FLAG
#GO 1
-->        0.000us FLAG                          ! The sequence of writes is
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us  ! as expected, but the times
VALUES ON WIRE-    1                             ! at which these writes occur
-->        0.000us FOR                           ! is incorrect. Also the secon
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us  ! write to data is 1111111110
VALUES ON WIRE-    1                             ! instead of 111111111100. Thu
-->        0.000us BAC                           ! it shows up in the simulatio
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us  ! run but is not caught by the
VALUES ON WIRE-    1                             ! simulator.
```

```
—>        0.001us FLAG
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.011us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.022us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.023us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.024us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.100us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
—>        0.100us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
—>        0.111us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.122us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.123us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.124us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.200us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
—>        0.200us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
—>        0.211us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.222us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.223us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.224us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.300us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
```

```
—>        0.300us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
—>        0.311us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.322us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.323us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.324us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.400us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
—>        0.400us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
—>        0.411us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.422us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.423us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.424us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.500us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
—>        0.500us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
—>        0.511us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.522us DATA
 LOGIC=   6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE—   11111111101
—>        0.523us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.524us BAC
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   0
—>        0.600us FOR
 LOGIC=   6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE—   1
```

```
-->        0.600us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
-->        0.611us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
-->        0.622us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
-->        0.623us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
-->        0.624us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
-->        0.700us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
-->        0.700us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
-->        0.711us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
-->        0.722us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
-->        0.723us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
-->        0.724us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
-->        0.800us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
-->        0.800us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
-->        0.811us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
-->        0.822us DATA
 LOGIC=    6 SIZE=  11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
-->        0.823us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
-->        0.824us BAC
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    0
-->        0.900us FOR
 LOGIC=    6 SIZE=   0 PERIOD=        0.000us
VALUES ON WIRE-    1
```

```
—>        0.900us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.911us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.922us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
—>        0.923us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.924us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        1.000us
#EXIT

6 garbage collection(s) in 236 ms

End of SIMULA program execution.
CPU time: 8.38  Elapsed time: 58.75
@pop

[PHOTO:  Recording terminated  Sat 29—May—82 5:23PM]
```

```
[PHOTO:  Recording initiated  Sat 29—May—82 5:33PM]

[Link from SPEAR, TTY 167]

 TOPS—20 Command processor 4(560)


! INJECTED ERROR IS A CONCURRENCY ERROR


@r concur
SLIDE/Multi—Level Simulator Version 1.0
Welcome and Good Luck!!

#GET TRAP.IL
#BAD COMMAND.
#BAD COMMAND.
#ALL
VISHAL: DEVA     DATA FOR BAC FLAG;
VITTAL: DEVB     DATA FOR BAC;
#SIMU
%Simulation time parameters for VISHAL : DEVA may be bound now
%finished
%Simulation time parameters for VITTAL : DEVB may be bound now
%finisned
```

```
#PR DATA
#PR F JOR
#PR BAC
#PR FLAG
#GO 1
—>        0.000us FLAG                                  ! The sequence of writes is as
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us         ! expected, but the times at
VALUES ON WIRE—    1                                    ! which they occur is incorrect
—>        0.000us FOR                                   ! The 'read/write' resource
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us         ! conflict is not caught by the
VALUES ON WIRE—    1                                    ! simulator. However, the
—>        0.000us BAC                                   ! omission of the NEXT
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us         ! statement causes a 'write/
VALUES ON WIRE—    1                                    ! write' resource conflict
—>        0.001us FLAG                                  ! which is caught by the
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us         ! simulator.
VALUES ON WIRE—    0
—>        0.011us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
%More than one write to B within one time instant        ! Error caught by the
—>        0.022us DATA                                  ! simulator.
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.023us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.024us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.100us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.100us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.111us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
%More than one write to B within one time instant
—>        0.122us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.123us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.124us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.200us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
```

```
—>        0.200us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.211us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
%More than one write to B within one time instant
—>        0.222us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    11111111100
—>        0.223us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.224us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.300us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.300us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.311us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
%More than one write to B within one time instant
—>        0.322us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    11111111100
—>        0.323us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.324us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.400us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.400us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    1
—>        0.411us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    11111111101
%More than one write to B within one time instant
—>        0.422us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE-    11111111100
—>        0.423us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
—>        0.424us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE-    0
```

```
—>        0.500us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.500us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.511us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
%More than one write to B within one time instant
—>        0.522us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.523us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.524us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.600us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.600us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.611us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
%More than one write to B within one time instant
—>        0.622us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.623us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.624us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.700us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.700us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.711us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
%More than one write to B within one time instant
—>        0.722us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.723us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
```

```
—>        0.724us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.800us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.800us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.811us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
%More than one write to b within one time instant
—>        0.822us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.823us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.824us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.900us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.900us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    1
—>        0.911us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111101
%More than one write to B within one time instant
—>        0.922us DATA
 LOGIC=    6 SIZE=   11 PERIOD=        0.000us
VALUES ON WIRE—    11111111100
—>        0.923us FOR
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        0.924us BAC
 LOGIC=    6 SIZE=    0 PERIOD=        0.000us
VALUES ON WIRE—    0
—>        1.000us
#EXIT

6 garbage collection(s) in 244 ms

End of SIMULA program execution.
CPU time: 8.34  Elapsed time: 58.08
@pop

[PHOTO:  Recording terminated  Sat 29—May—82 5:34PM]
```

APPENDIX C

The Relationship Between

HARDWARE SYNTHESIS AND HARDWARE VERIFICATION

Table of Contents

List of Figures

1. ABSTRACT

This paper presents a discussion of the relationship between hardware synthesis and hardware verification. Some examples of synthesis are presented and their relationships to verification are explored. Finally, a particular synthesis method is singled out and shown to be applicable to verification.

2. INTRODUCTION

The fields of hardware synthesis and verification are relatively young. A survey of the verification field shows that less than a dozen widely read publications in hardware verification existed prior to 1978. While hardware synthesis work has been more widely published, the theory is less developed, since there is no directly analogous research which can be built upon.

There is a need to develop more formal models of the design process if synthesis theory is to progress. Therefore, if there were to be a real relationship between the models and techniques used in hardware verification and their analogous synthesis problems, both areas of research could build on each other. Furthermore, work in synthesis could provide empirical validation of the formalisms used in verification, and the goal-directed approach used in synthesis might be applicable to verification as well.

Before we begin the argument for a strong relationship between verification and synthesis, however, we should point out that the groundwork has already been laid for us. As we shall see, there are parts of the synthesis process which are equivalent to steps in the verification process and vice-versa. Thus, the major thrust of the paper is not in revealing that the relationships exist, but in showing where this equivalence lies. In order to do that, we turn to experimental work in hardware synthesis, and show how each example can be related to a corresponding verification problem. After doing this, we present some examples of verification research which bear resemblance to analogous synthesis problems. Finally, the paper gives an example synthesis

technique and shows how the technique can be redirected to perform verification instead.

3. RELATED RESEARCH

There have been some key papers which have shown a clear relationship between verification and synthesis. An example paper is the work by Roth [11] where he describes the synthesis of combinational logic from Boolean expressions. The intent of the synthesis work is to provide a basis of comparison to manual designs in order to verify that the human designs are logically correct. Thus, synthesis in this case is just a step in the process of proof of equivalence.

A second paper [1] describes the synthesis of part of the control flow of a machine, given a high-level specification of the desired action, and the operations available in the data paths. The synthesis is carried out by parsing the high level specification of the control, using production rules derived from the data path description. In essence, the paper is describing a technique which can be used to verify that a desired action can be executed by a given machine, via an existence proof. Although the technique described is not a complete verification because parallel cases have not been considered, it does illustrate the relationship.

In a related domain, a comparison of microcode synthesis and microcode verification has been published [9]. In this paper, the idea is proposed that automated synthesis can be viewed as a theorem-proving process, with the resulting microcode a side-effect of verification of the high-level specification.

With these efforts as somewhat obvious examples of synthesis- verification relationships, we now proceed to define the fundamental steps involved in synthesis and verification.

4. THE SYNTHESIS AND VERIFICATION PROCESSES

Before we discuss hardware synthesis and verification, we need to clarify the terms synthesis and verification. In doing this, we will examine the processes that go on during verification and synthesis tasks.

Synthesis

Synthesis of digital hardware involves two major tasks:

- ALLOCATION: Mapping (binding) of abstract characteristics of the behavior of a digital design to specific elements of the design structure. Mathematically, variables representing design alternatives or design variations become constants.
- OPTIMIZATION: Transformations on a design at some stage in the design process which do not change the level of the design or add more design detail, but which tend to maximize or minimize some attribute of the design such as cost or speed by changing the existing design. Mathematically, constant values or structures representing design features are changed during optimization.

In reality, OPTIMIZATION and ALLOCATION are often mixed in a single synthesis procedure. We separate them here for the purposes of the discussion.

ALLOCATION is goal-directed, and one or more design objectives such as silicon area, critical path delays, gate count, reliability, testability, and/or average performance determine the way in which allocation is performed on a particular design.

Binding, or ALLOCATION of hardware structures to a behavioral specification involves three types of actions:

1. Determination of required resources to implement the specified behavior.
2. Scheduling of functions onto the resources, thereby determining the ordering of operations.
3. Composition of available resources to implement a required function when no one-to-one mapping from function to structure exists.

In reality, actions 1 and 2 are closely related, and optimal solutions to synthesis are only guaranteed when both problems are solved simultaneously. An example of a type 1 action is the decision to require the hardware to contain an adder and three registers, and to require interconnection pathways so that the results of an add can be saved in any of the three registers. Type 2 scheduling decisions involving this structure could include storing variable A in register 1 until it is no longer needed, then using register 1 to store variable Z, or doing the add of variables A and B before the add of A and C, thus scheduling the adder. Type 3 actions are exemplified by the construction of structures from more primitive elements to meet functional requirements. For example, a logical memory addressing scheme might be implemented by building up a physical memory addressing scheme out of available memory elements. Another example is the construction of an "n" bit register out of flip-flop and gate standard cells, because no single design which implements an "n" bit register exists.

Now let us describe the allocation process somewhat more formally. Entities E which exist in the abstract are assigned (allocated) to resources R used to implement each entity. The assignment function A performs a mapping from entity e to entity-resource pair er:

$$A(e) \longrightarrow er | er \in \{E \times R\}$$

Partial orderings between entities are mapped onto specific locations, either in the xy plane or along the time axis. For example, an entity e is mapped into an entity-position pair $(e, P_{x,y})$.

Requirements on the resource (such as propagation delays, area consumption and minimum bit widths) become the actual capabilities of the resource. In general, a range of allowed values VA maps into a range of measured values VM.

Thus, ALLOCATION can be expressed as the following mapping

ALLOC (e,VA) —> er, $P_{x,y}$, VM| er ε {E X R}

In addition, a resource used to implement an entity may be composed of simpler, preexisting resources as part of the allocation process. (This final step can be performed either by composing the resource out of simpler resources or by decomposing the entity into simpler entities, each of which maps into a single resource.)

Some examples of entity-resource pairs are:

1. Values, mapped onto registers
2. Two-input NAND functions, mapped onto three transistors, one of which performs a pullup function.
3. A Boolean equation set, mapped onto a PLA
4. An ADD function, mapped onto an arithmetic-logic unit(ALU)

The first example also contains a mapping from a partial ordering in time (between which the variable is produced until it is no longer needed) to a specific time period when the register contains the value. In this example, the requirement of storing an 8-bit value becomes the actual capability of the register to store values up to 8 bits in width.

The second example illustrates a composition of a resource implementing the NAND function from a collection of functional and pullup transistors. This example could also contain the allocation of xy positions to each transistor from the partial ordering of positions required by the transistor interconnections and the location of the NAND function in the surrounding circuitry.

The third example illustrates the many-to-many mapping of entities to resources inherent in synthesis. The fourth example illustrates an entity-resource mapping where the function provided by the resource to implement the entity is one of a set of functions provided by the resource.

Depending on the synthesis task, the three allocation phases(the allocation of entity to resource, the mapping of partial order to complete order, and the composition of complex resources from simpler ones) may all be performed simultaneously, may be performed in a predefined order, or may not be performed at all.

OPTIMIZATION, on the other hand, is a process where existing entity-resource mappings are altered, complete orderings are rearranged, and/or resource parameters are changed.

A major part of the OPTIMIZATION task is deciding which transforms are appropriate for a given problem, and in what order they should be applied. Designs are transformed in order to optimize one or more design objectives, or goals. The selection of transformations and their ordering is done with an eye to one or more of these goals. It is important that the transformations preserve the required (correct) behavior of the hardware, while altering the underlying structure. Therefore, a preservation of correct behavior is contained in the OPTIMIZATION task. However in most systems this is implicit in the transformations rather than being stated formally.

One example of a transformation in the high-level specification of a design is the substitution of a constant result for a calculation involving two constants as operands. Translation of an n-level array of combinational logic into two-level logic is an example of a transformation performed at a lower level in the design process. A third example is iterative improvement of standard cell placement by pairwise interchange.

Verification

Hardware verification, like software verification, is involved in proving the equivalence of two objects. We should point out that most notions of hardware equivalence differ from those of software equivalence. The primary difference seems to be in requiring certain sequential behavior from hardware

that is not required from software. See [8] and [10] for discussions and examples of this.

Hardware verification consists of two major steps. They are

- LEVELING: This involves changing the level of a design specification either by performing allocation or by abstracting out some design aspects via the composition process. This step is done in order to obtain two design specifications at the same level which can then be compared. Of course, if the descriptions already exist at the same level, this step is omitted.
- COMPARISON: This step involves transforming one or both design specifications according to certain rules (axioms) until they are proven to be equivalent.

Note that the above steps can involve composition, allocation, and transformations on the design, just as synthesis does. However, allocation and composition are done here to produce a single candidate design, however far from optimal it is. Transformations are performed with a specific end product in mind, rather than to attain overall design objectives. Thus, the relationship between hardware synthesis and hardware verification can be clarified. The mechanisms used in performing synthesis and verification are identical in many cases. These mechanisms involve the manipulation of design descriptions. However, the policy which determines the choice of mechanisms and the order in which they are to be applied is different, and the policy which assesses the result of each application is different, depending on whether the end goal is synthesis or verification.[1]

Examples

We now present some synthesis and verification examples which demonstrate the relationship we have been discussing. An example of allocation is the

---

[1]This distinction between policy and mechanism is borrowed from operating system terminology.

research performed by Hafer and Parker [2], [3]. Figure 4-0, adapted from [4], shows an entity-resource mapping and a partial ordering to specific time mapping.

In the second example, we discuss the generation of control using LCD [1]. Here, we have a high-level statement to form the sum of IAR and 8 bits of the instruction register padded with higher-order zeros. The statement is

ACC:= SUM(IAR,'00000000'||INST/8:15/)

The actual control sequences generated to execute this statement are

- Load INST onto DBUS and Load B from DBUS
- Load IAR onto DBUS and Load A from DBUS
- Set ALU function inputs to ADD and Load ACC from DBUS

In essence, a single operation in time gets implemented by three distinct steps, ordered in time. The SUM operation gets allocated onto the ALU. The transfer operation, implicit in the higher-level statement, gets mapped onto explicit transfers on and off the DBUS. Composition of two transfers (on and off the bus) in order to implement the implicit transfer also occurs.

Another example of composition is found in the synthesis work by Leive [6], shown in Figure 4-1, taken from [6]. The figure shows the stepwise partitioning of a register-transfer design in order to compose the register and adder from more primitive elements.

Examples of transformations are found in the research by Snow and McFarland [12] and [8], and the research by Wagner [13]. The research by Snow describes optimizing transformations on a high-level data-flow description called the Value Trace . McFarland has proven that these transformations preserve abstract behavior. One particular transformation removes redundant operators, as shown by the removal of the multiply operator in Figure 4-2, adapted from [7].

Figure 4-1: Example of Entities-Resources and Timing Mappings

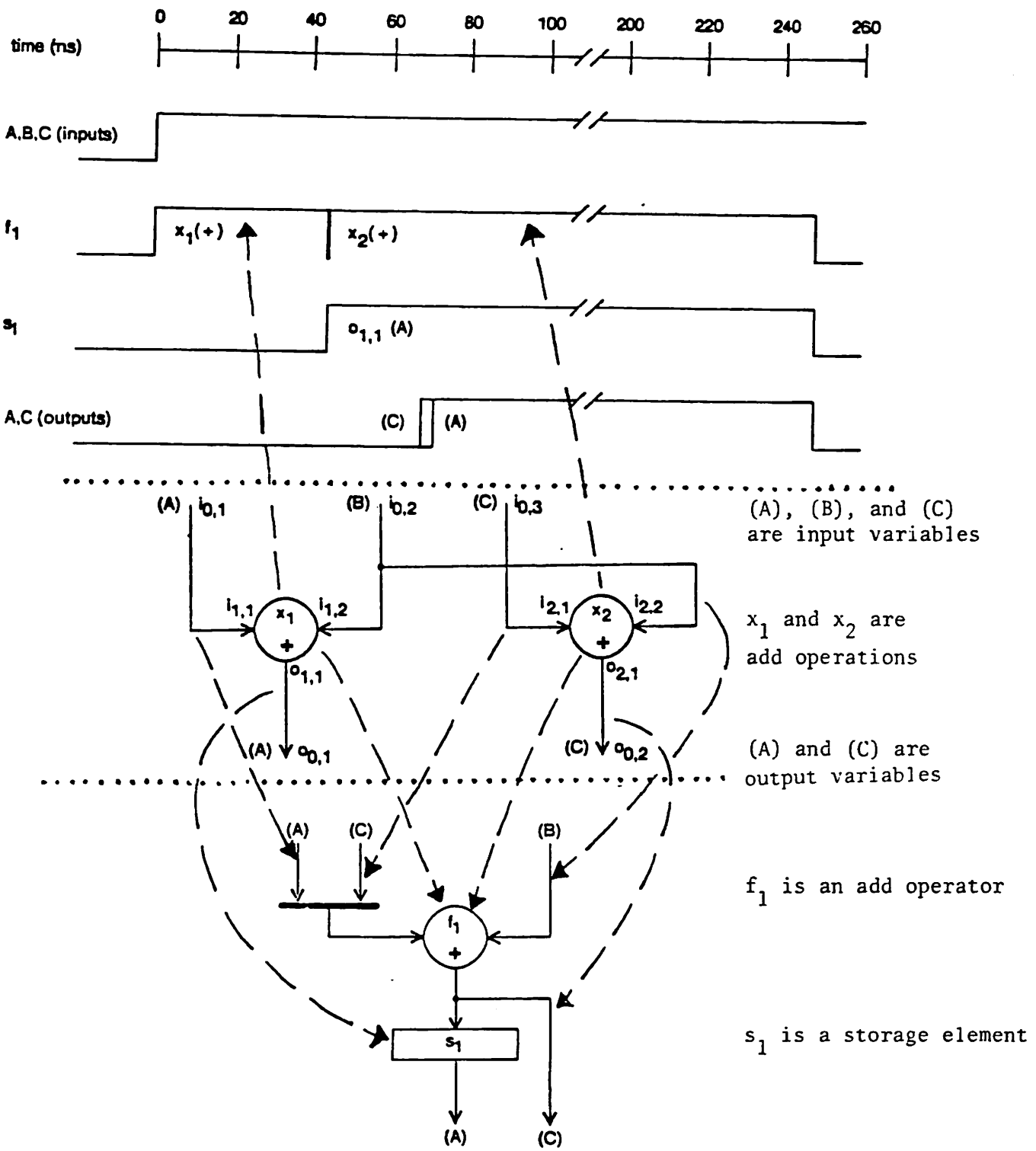


Figure 4-2: Partitioning of Register-Transfer Design

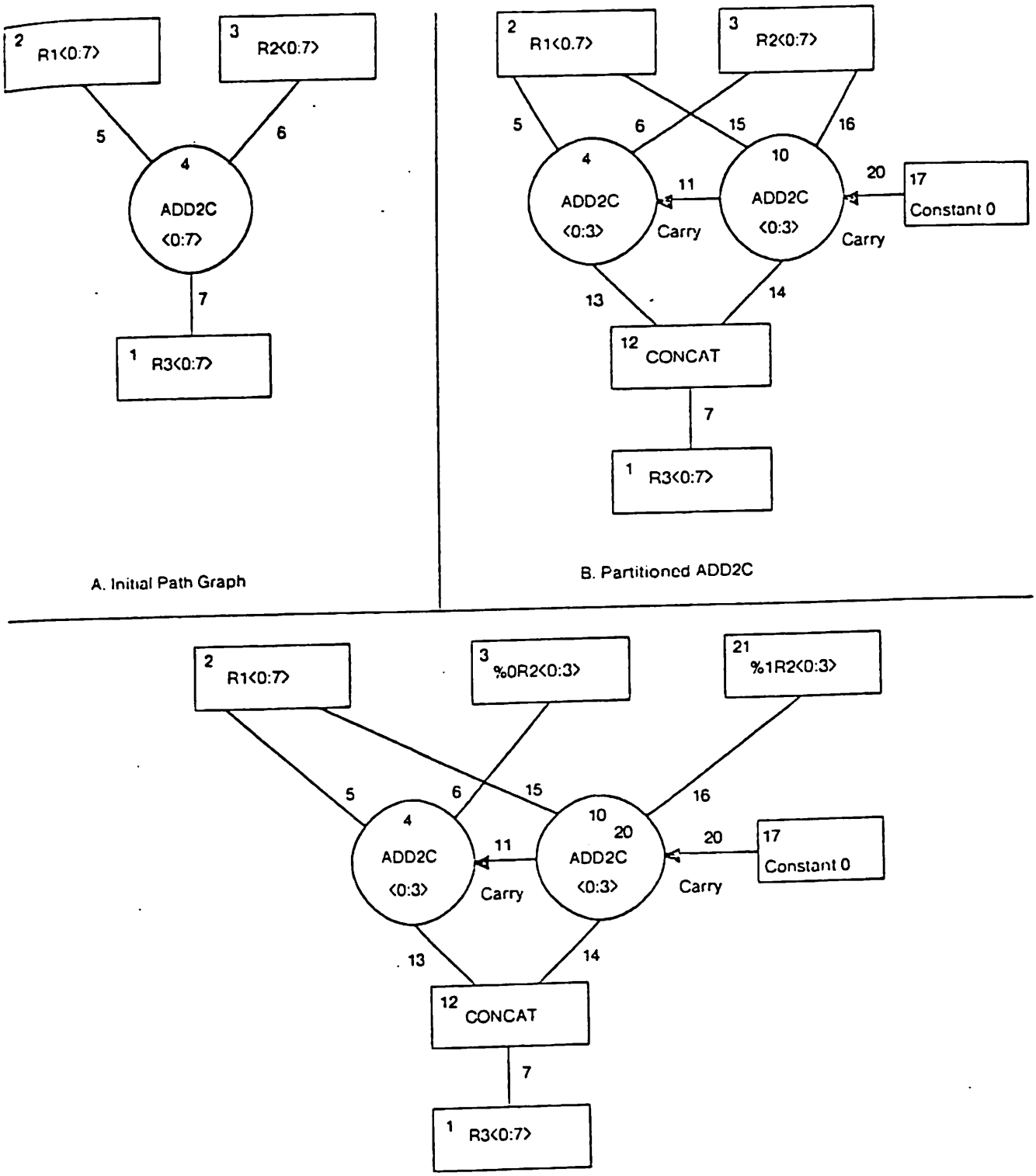
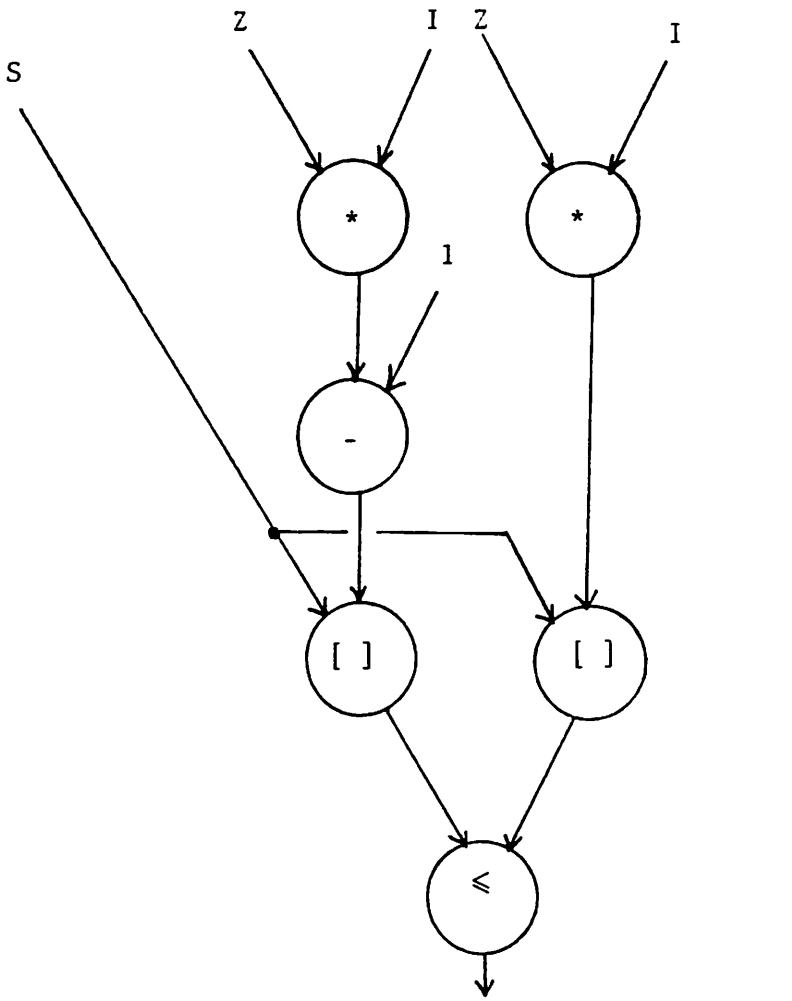
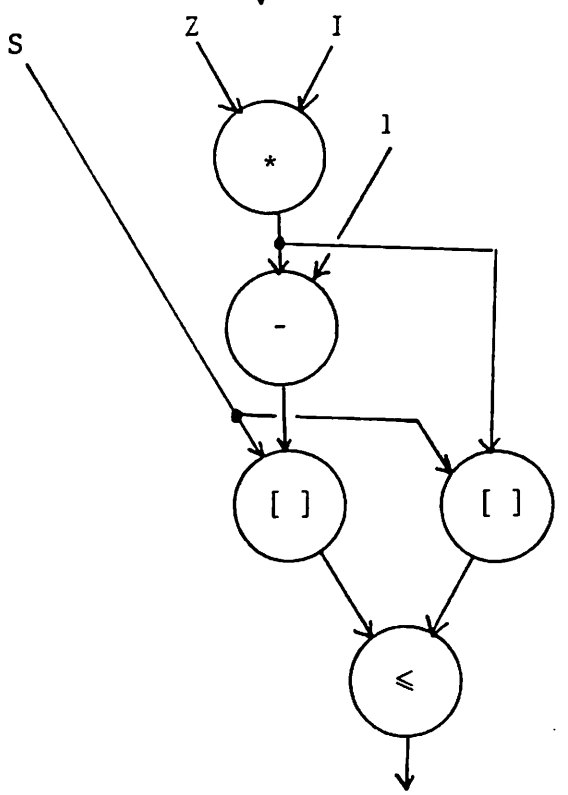


Figure 4-3: An Original and a Transformed Value Trace



Original Value Trace

If X[Z*I-1] ≤ S[Z*I]



Value Trace After Removal of Redundant Multiply

Wagner transforms a hardware description according to certain axioms in order to prove the truth of assertions about the description. An example of this is the transformation from

/~CLEAR/ Q[0]<-0 ; /~CLEAR/ Q[1]<--0;

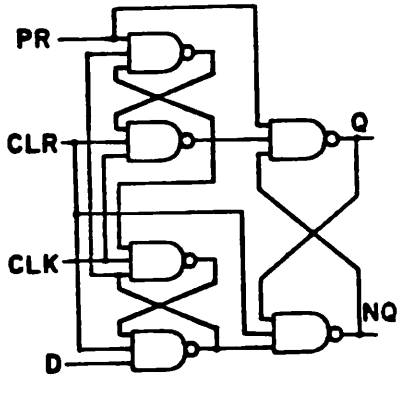
to

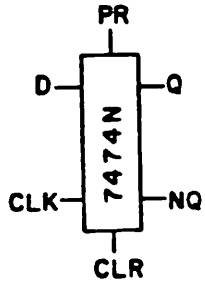
/~CLEAR/ Q[0:1]<-(0&0);

where & indicates concatenation.

Leveling by abstracting behavior from structure is performed by Leinwand and Lamdan [5] in order to perform verification. They extract a behavior, in terms of /condition/action statements, from a gate-level structure. An example taken from [5] is shown in 4-3.

flip - flop type "7474N".






```
-----------------------------------------
Structural Definition of Element 7474N
-----------------------------------------
Output Interfaces : Q NQ
-----------------------------------------
Input Interfaces : D CLK PR CLR
-----------------------------------------
Interconnections :
  7410N      #7256  #7254   CLK   #7255
  7410N      #7255  #7256   CLR     D
  7410N      #7257   PR    #7255  #7254
  7410N      #7254  #7257   CLR    CLK
  7410N       Q     NQ     PR    #7254
  7410N       NQ    Q      CLR   #7256
-----------------------------------------
-----------------------------------------
Functional Definition of Element 7474N
-----------------------------------------
Output Interfaces : Q NQ
Input Interfaces  : D CLK PR CLR
-----------------------------------------
Behavior :
/ *PR' + PR . *CLR' + CLR' . *PR + CLR . PR .
*CLK /
  Q (-- *PR' + D . CLR . PR . *CLK
/ *CLR' + CLR . *PR' + PR' . *CLR + CLR . PR .
*CLK /
  NQ (-- *CLR' + D' . CLR . PR . *CLK
-----------------------------------------
```

Figure 4-4: Example of Leveling by Behavior Extraction

5. A SPECIFIC EXAMPLE: THE APPLICATION OF A SYNTHESIS MODEL TO VERIFICATION

The synthesis of register-transfer structure from register-transfer behavior can be performed by expressing the required behavior as a set of algebraic constraints on the design [4], [2]. These constraints are of two types:

1. Rules about the mapping from specific aspects of RT behavior to RT structure, including allocation of specific times to events.
2. General rules about how structures can be used to implement behavior.

Two examples of these rule types follow.

$$T_{OA}(O_a) = T_{XS}(x_a) + \sum \sigma_{d,a} D_{FP}(f_d)$$

$$\sum \sigma_{d,a2} = 1 \qquad d|f_{da} \in F_a$$

$$d|f_{da} \in F_{a2}$$

The first equation specifies that the time $T_{OA}(O_A)$ when the outputs O_a of operation x_a will be available is the time the operation began, $T_{XS}(x_a)$ and the propagation time through the operator, D_{FP}. Now, since we do not know which operator will actually be allocated to the operation x_a, we use the summation as a selection operation. $\sigma_{d,a}$ is a 0-1 variable which is set to one to indicate that operator f_d is used to implement operation x_a. Thus, the σ variable selects the proper propagation delay.

The second equation is a general one about design practices. It states that one and only one operator must be selected to implement an operation.

Now, we pose two verification problems - one straightforward, the other a little more complicated. First, suppose we wish to verify that a given set of data paths can correctly execute a required behavior, if we know the entity-

resource mapping. In other words, we want to verify here that the data paths and the entity-resource mapping specified can be used to execute the specified behavior and that the timing is correct. Knowing the entity-resource mapping is equivalent to knowing the values of the Greek variables which are used to express design alternatives. These variables become constants in the constraints and we can solve the constraint set using linear programming. Once we find a feasible solution, or find that none exists, the timing verification is done.

The second verification problem is the following: Given a set of data paths, and a specification of the required register- transfer behavior, prove that some entity-resource mapping exists so that the data paths can execute the required behavior correctly. In specific, what we would like to prove is that there are sufficient storage and operator resources (as well as interconnections between these) to map the required behavior onto the structure. We indicate the data path structure by setting up the same mixed 0-1 linear program described above and restricting the set of available operators and storage elements the software can choose to those actually available in the data paths.

In addition, we need new 0-1 variables specifying the pathways available between registers and operators. We introduce these variables for a number of reasons. One reason is that we only want to consider storing variables where they will be accessible to operators which use them as inputs. (These variables do not exist in the current synthesis model because pathways are not explicitly allocated; one can deduce their presence from the location of variables in registers). These variables are

- $\alpha_{e,c,a2,b}$:A path exists from the output $o_{e,c}$ of a register S_e to one of the inputs $i_{a2,b}$ of an operator x_{a2}
- $\beta_{a1,c,a2,b}$:A path exists from the output $o_{a1,c}$ of an operator x_{a1} to one of the inputs $i_{a2,b}$ of another operator x_{a2}

- $\eta_{a1,c1,e,c2}$: A path exists from the output $o_{a1,c1}$ of an operator x_{a1} to the input $i_{e,c2}$ of a register S_e

We use these variables to create additional constraints which force proper use of the data path interconnections between operators and registers. In essence, these constraints state

- The output $o_{a,c}$ from an operator x_a can only be stored in a register S_e when $\eta_{a,c,e,ci}$ is true for some input ci of the register.
- The output $o_{a1,c}$ from an operator x_{a1} can only be used directly by another operator x_{a2} as input $i_{a2,b}$ if $\beta_{e,c,a,b}$ is true.
- The output $o_{c,e}$ of a register S_e can only be used by an operator x_a as input $i_{a,b}$ if $\alpha_{e,c,a,b}$ is true.

Solution to these constraints along with the synthesis constraints from [2] provides a verification that the data paths can support the required behavior. If no feasible solution exists, then the data paths do not support the required behavior. Thus, we have shown here that a synthesis technique can be modified and applied to the verification process.

6. THE IMPORTANCE OF THE SYNTHESIS-VERIFICATION RELATIONSHIP

We will now address the importance of the synthesis-verification relationship. The fact that the mechanisms involved in synthesis and verification can be shown to be identical in many cases is important for three reasons. First of all, proofs of equivalence can use heuristics to select transformations based on knowledge about overall design constraints and objectives. For example, to prove that a certain register-transfer structure implements a specified register-transfer behavior, the following steps can be performed. The behavior can be allocated into a candidate structure (LEVELING) and then transformed selectively (COMPARISON) until it is shown to be equivalent to the given structure. The transformation process can be undertaken with the knowledge that the existing design has been done under certain constraints, limiting the search space.

Second, formal models and formal semantic definitions produced as part of the verification process become equally applicable to synthesis problems. This underlying common formalism can then be exploited in the construction of CAD systems. Thus, the third reason why recognition of the synthesis-verification relationship is important is the insight it gives to CAD system construction. It encourages the separation of policy from mechanism, the reuse of mechanisms for different applications, and supports interactive (human supplies the policy) as well as automatic synthesis and verification.

7. ACKNOWLEDGMENTS

The author wishes to acknowledge many fruitful discussions about this topic with Mike McFarland, Steve Crocker, and others which led to this paper.

REFERENCES

[1] Evangelisti, C. J., Goertzel, G., and Ofek, H.
Using the Dataflow Analyzer on LCD Descriptions of Machines to Generate Control.
In Proceedings of the 4th International Symposium on Computer Hardware Description Languages. ACM and IEEE Computer Society, October, 1979.

[2] Hafer, L., Parker, A.
A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic.
In Design Automation Conference Proceedings no. 18, pages 846-853. ACM SIGDA IEEE Computer Society-DATC, June, 1981.

[3] Hafer, L., Parker, A.
Automated Synthesis of Digital Hardware.
IEEE Transactions on Computers C-31(2):93-109, February, 1981 .

[4] Hafer, L.
Automated Data-Memory Synthesis : A Formal Model for the Specification, Analysis and Design of Register-Transfer Level Digital Logic.
PhD thesis, Dept of Electrical Engineering, Carnegie Mellon University, Pittsburgh, Pa., May, 1981.

[5] Leinwand, S. and Lamdan, T.
Design Verification Based on Functional Abstraction.
In Proceedings of the 16th Design Automation Conference, pages 353-359. ACM SIGDA and IEEE Computer Society DATC, June, 1979.

[6] Leive, G. W.
The Design, Implementation, and Analysis of an Automated Logic Synthesis and Module Selection System.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, January, 1981.

[7] McFarland,M.
The Value Trace: A Data Base for Automated Digital Design.
Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., December, 1978.

[8] McFarland, M.
Mathematical Models for Verification in a Design Automation System.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, July, 1981.

[9] Mueller, R. A., and Johnson, G. R.
Contrasting Translation, Verification and Synthesis in Software and Firmware Engineering.
In Proceedings of the 14th Microprogramming Workshop, pages 17-22. October, 1981.

REFERENCES

[1] Evangelisti, C. J., Goertzel, G., and Ofek, H.
Using the Dataflow Analyzer on LCD Descriptions of Machines to Generate Control.
In Proceedings of the 4th International Symposium on Computer Hardware Description Languages. ACM and IEEE Computer Society, October, 1979.

[2] Hafer, L., Parker, A.
A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic.
In Design Automation Conference Proceedings no. 18, pages 846-853. ACM SIGDA IEEE Computer Society-DATC, June, 1981.

[3] Hafer, L., Parker, A.
Automated Synthesis of Digital Hardware.
IEEE Transactions on Computers C-31(2):93-109, February, 1981 .

[4] Hafer, L.
Automated Data-Memory Synthesis : A Formal Model for the Specification, Analysis and Design of Register-Transfer Level Digital Logic.
PhD thesis, Dept of Electrical Engineering, Carnegie Mellon University, Pittsburgh, Pa., May, 1981.

[5] Leinwand, S. and Lamdan, T.
Design Verification Based on Functional Abstraction.
In Proceedings of the 16th Design Automation Conference, pages 353-359. ACM SIGDA and IEEE Computer Society DATC, June, 1979.

[6] Leive, G. W.
The Design, Implementation, and Analysis of an Automated Logic Synthesis and Module Selection System.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, January, 1981.

[7] McFarland,M.
The Value Trace: A Data Base for Automated Digital Design.
Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., December, 1978.

[8] McFarland, M.
Mathematical Models for Verification in a Design Automation System.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, July, 1981.

[9] Mueller, R. A., and Johnson, G. R.
Contrasting Translation, Verification and Synthesis in Software and Firmware Engineering.
In Proceedings of the 14th Microprogramming Workshop, pages 17-22. October, 1981.

Second, formal models and formal semantic definitions produced as part of the verification process become equally applicable to synthesis problems. This underlying common formalism can then be exploited in the construction of CAD systems. Thus, the third reason why recognition of the synthesis-verification relationship is important is the insight it gives to CAD system construction. It encourages the separation of policy from mechanism, the reuse of mechanisms for different applications, and supports interactive (human supplies the policy) as well as automatic synthesis and verification.

7. ACKNOWLEDGMENTS

The author wishes to acknowledge many fruitful discussions about this topic with Mike McFarland, Steve Crocker, and others which led to this paper.

[10] Overman, W. T.
Verification of Concurrent Systems: Function and Timing.
PhD thesis, Computer Science Department, University of California, Los Angeles, August, 1981.

[11] Roth, J. P.
Hardware Verification.
IEEE Transactions on Computers C-26(12):1292-1293, December, 1977.

[12] Snow,E.
Automation of Module Set Independent Register Transfer Level Design.
PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April, 1978.

[13] Wagner, T.
Hardware Verification.
PhD thesis, Department of Computer Science, Stanford University, 1977.

APPENDIX D

Proofs of Atomic Action

13. APPENDIX D

PROOFS OF ATOMIC ACTIONS

We shall first consider the atomic actions and then use the results of these proofs for the proofs of the complex actions.

A. Assignment to an Internal Variable

u <— e

The above atomic action assigns the value of expression e to the local variable u. The errors which are possible are that the atomic action could be missing, the value assigned to u could be incorrect, the variable to which the assignment occurs is wrong or the position in which the assignment occurs is wrong. The last possibility shall be considered separately since it forms a different and general class of errors, i.e., wrong sequencing of events.

Consider the Behavior Expression of an arbitrary program which has u<—e as one of its statements. (The BE of u <— e is λ:/T[u<—e](P)/, where λ is the empty event):

$$B^1.\lambda/T[u<\!-\!-e](P)/.B^2.B^3 \tag{7}$$

where B^1 and B^3 are some complex and arbitrary Behavior Expressions and B^2 is a Behavior Expression corresponding to any of the atomic or complex actions listed in the previous section.

We have assumed that the program which is represented by the expression listed in 7 has the assignment as an intermediate statement. The other possibilities are that the assignment could be either the first or last statement of the program, in which case the Behavior Expression would be either

$$\lambda:/T[u <\!-\!- e](P)/.B^1.B^2 \tag{8}$$

or

$$B^1.\lambda:/T[u \leftarrow\!- e](P)/ \tag{9}$$

The expression in 9 is equivalent to B^1 using λ-reduction [4] and thus any error in the assignment is not going to affect the Behavior Expression. This means that if an assignment to a local variable is the last action in a hardware description, it does not affect the externally visible behavior. This is as expected, since local variables are only observed when they affect global variables.

Thus we have to consider 7 and 8 only. The proofs for both of them are the same and thus we shall consider the expression in 7.

Case (i:Missing atomic action)

We shall consider what happens to the behavior expression if the assignment u <— e is missing. We will have to consider that the assignment could be followed by any of the atomic or complex actions. Thus we shall consider each case separately.

Case (a): Assignment followed by x_i <— e. In this case the Behavior Expression would be

$$B^1.\lambda:/T[u \leftarrow\!- e](P_1)/.\ \ W(X_i):/P_2 \wedge x_i = e/.B^3 \tag{10}$$

$/T[u \leftarrow\!- e](P_1)/$ is the postcondition of the assignment u <— e and P_1 is the precondition of the assignment. P_2 is the precondition of the statement x_i <— e.

Now $P_2 = /T[u \leftarrow\!- e](P_1)/$; i.e., the postcondition of the assignment u <— e is equal to the precondition of the assignment x_i <— e.

Now if the assignment is missing, P_2 is going to be incorrect, unless $u' \neq u$, where u' = value of the variable after assignment and u the value of the variable before the assignment. This special case is the only one where a missing assignment to a local variable cannot be detected by a change in the BE.

Thus P_2 will be changed to P_2' where $P_2 = /T[u \leftarrow e](P_1)/$ and $P_2' = /P_1/$. Hence the Behavior Expression will be

$$B^1.\ \ W(x_i):/P_2' \wedge x_i = e/.\ \ B^3 \tag{11}$$

instead of

$$B^1.\ \ W(x_i):/P_2 \wedge x_i = e/.\ \ B^3 \tag{12}$$

The expression in 10 reduces to the expression in 12 by λ-reduction. Hence the missing atomic action is going to manifest itself in the predicate.

Case (b): Assignment followed by $u \leftarrow x_i$.

In this case the Behavior Expression would be

$$B^1.\ \lambda :/T[u \leftarrow e](P_1)/\ .\ \ R(x_i):/T[A](P_2)/.B^3$$

The argument is exactly the same as in Case (a). Thus the Behavior Expression would be modified (unless $u = u'$) to

$$B^1.\ R(x_i)\ :\ /\ T[A](P_2')/.\ B^3$$

Thus the error would manifest itself in the predicate of the write action.

Case (c): assignment followed by "leave f_k".

In this case the situation would be as follows, since leave f_k has to be the last statement of a procedure, unless it is part of a complex action:

Procedure
.
.
.
.
.
u <— e
leave f_k

Any procedure call maps into a loop with a loop counter. The Behavior Expression in this case would be as shown below:

$$B^1.[B^2.\lambda:/T[u\langle\!-\!-e](P_1)/.\lambda:/P_2/]^*I_k.B^3.B^4$$

where B^1, B^2, B^4 are complex arbitrary expressions and B^3 is an expression corresponding to any of the atomic or complex actions.

$$P_2 = /T[u\langle\!-\!-e](P_1)/$$

Unless $u=u'$ P_2 will be modified to P_2'. Thus $L[A,f_k]$ (P_2) will be modified to $L'[A,f_k](P_2')$.

Now the correct post condition to the call of the procedure is

$$\exists\, I\ \{L[A,f_k](P_k(P,I)) \vee T[A_k](P_k(P,I))\} \qquad (13)$$

Thus the postcondition will now be modified. The postcondition is the precondition for the next action corresponding to B^3. Thus the error will show up in the predicate of B^3.

Case (d): Assignment followed by restart f_k. Again "restart f_k" will be the last statement of the procedure. The Behavior Expression is going to be

$$B^1.\ [B^2.\lambda:/T[u\langle\!-\!-e](P_1)/.\lambda:/P_2/]^*I_k.B^3.\ B^4$$

Where B^1, B^2, B^4 are arbitrary complex expressions and B^3 is an expression corresponding to any of the atomic or complex actions.

$$P_2 = /T[u\langle\!-\!-e](P_1)/$$

If $u \neq u'$ P_2 is replaced by P_2'. Thus

$$R[A, f_k](P_2)$$

will be replaced by

$$R'[A, f_k](P_2')$$

Thus equation 13 is going to be modified because

$$P_k\ (P, I)$$

is modified to

$$P'_k(P, I)$$

since now

$$P_k\ (P, I) = ((I=1 \wedge P) \vee ((I>1 \wedge R'[A, f_k](P_k(P, I-1)))$$

Hence the postcondition to the call of the procedure is modified and this will show up in the predicate of B^3 as discussed in Case c.

Case (e): Assignment followed by 'if b then A_1 else A_2'. The Behavior Expression is

$$B^1.\lambda:/T[u\leftarrow e](P_1)/.[\ B[A_1](P_2 \wedge b) + B[A_2](P_2 \wedge \sim b)].\ B^3.B^4$$

if $u \neq u'$ then P_2 is replaced by P_2' and the error will show up in the predicate.

Case (f): Assignment followed by A_1; A_2. The BE is

$$B^1.\ \lambda/T[u\leftarrow e](P_1)/.B[A_1](P_2)\,||\,B[A_2](P_2).B^3.B^4$$

The argument is the same as Case e.

Case (g): Assignment followed by A_1 next A_2. The BE is

$$B^1.\lambda:/T[u\leftarrow e](P_1)/.\ \ B[A_1](P_2).B[A_2]/(T[A_1](P_2)\ /.\ \ B^3\ .\ \ B^4$$

The argument is the same as in Case e.

Case (h): Assignment followed by call f_k. The BE is

$$B^1.\lambda:/T[u\leftarrow e](P_1)/.[B[A_k](P_k(P_2,I_k))]^*I_k \ . \ B^3 \ . \ B^4.$$

The argument is the same as in Case e.

Case (ii): The value assigned to u is incorrect: ($u\leftarrow e'$ instead of $u\leftarrow e$). In this case $T[u\leftarrow e'](P_1)$ will be different from $T[u\leftarrow e](P_1)$ which implies P_2 is changed to P_2'. Thus removing the condition if $u \equiv u'$ (never possible in this case) the proof is the same as in Case (i) for cases (a) through (h).

Case (iii): The variable to which the value is assigned is incorrect: $u_2 \leftarrow e$ instead of $u_1\leftarrow e$. In this case unless $u_2 \equiv u_2'$ and $u_1 \equiv u_1'$ the error will show up and the proof is the same as in Case (i) for cases (a) through (h).

B. Assignment to an External Variable

$x_i\leftarrow e$.

Consider the Behavior Expression of a program which has $x_i \leftarrow e$ as one of its statements

$$B^1 \ . \ W(x_i) \ : \ /P\wedge x_i = e \ / \ . \ B^2 \qquad (14)$$

where B^1, B^2 are some arbitrary complex BE's. The errors which are possible are listed below.

(i) missing atomic action, $x_i \leftarrow e$.

(ii) incorrect atomic action, $x_i \leftarrow e'$ instead of $x_i\leftarrow e$.

(iii) variable to which assignment is made is incorrect, $x_i'\leftarrow e$ instead of $x_i\leftarrow e$.

(iv) read or λ instead of write.

Case (i): Missing atomic action. The BE in 14 will be modified to

$$B^1 \,.\, B^2 \tag{15}$$

Thus the error will manifest itself as a missing event schema in the BE.

Case (ii): Incorrect atomic action. The BE in 14 will be modified to

$$B^1.\ W(x_i) : /P \wedge x_i = e'/ \,.\, B^2$$

Thus the error manifests itself as an error in the predicate.

Case (iii) Incorrect variable. The BE in 14 is modified to

$$B^1 \,.\, W(x_i'): /(P \wedge x_i' = e)/ \,.\, B^2$$

Thus the error manifests itself as an error in the predicate.

Case iv: Wrong action. The BE in eq. 14 is modified to

$$B^1.R(x_i):/P/.B^2$$

or

$$B^1.\lambda:/T[u \leftarrow\!\!- e]P/.B^2$$

depending on whether the wrong action is a read or λ.

C. Read From an External Variable

$$u \leftarrow\!\!- x_i$$

Consider the BE of a program which has $u \leftarrow\!\!- x_i$ as one of its statements

$$B^1 \,.\, R(x_i) : / T[A](P_1) / \,.\, B^2 \,.\, B^3$$

where B^1, B^3 are arbitrary complex expressions and B^2 is a BE corresponding to any of the atomic or complex actions.

The possible errors are:

(i) wrong global variable is read

(ii) atomic action is missing

(iii) the value is assigned to the wrong variable

(iv) write or λ instead of read

Case (i) Wrong global variable is read and the BE is

$$B^1.\ R(x_i'):\ /\ T[u\langle\!-\!-x_i'](P_1)/\ .\ B^2\ .\ B^3$$

Thus the BE is modified (x_i becomes x_i') and $T[u\langle\!-\!-x_i](P_1)$ becomes $T[u\langle\!-\!-x_i'](P_1)$. Thus the predicate is also changed since the postcondition $T[u\langle\!-\!-x_i](P_1)$ is going to be different unless $x_i \equiv x_i'$ (in which case the error cannot be detected).

Case (ii) Missing atomic action. The BE is

$$B^1\ .\ B^2\ .\ B^3$$

An event schema is missing from the BE, and the detection of the error is guaranteed.

Case (iii) Variable u is incorrect, i.e., u_2 instead of u_1. Now if $u_2' \equiv u_2$ and $u_1' \equiv u_1$ then the internal state remains unchanged and the BE is the same. However, if the above condition is not satisfied then $T[u\langle\!-\!-x_i]$ (P) will be modified, the predicate will be modified, and the error is detectable.

Case (iv) Wrong action. The proof is similar to B, Case (iv).

D. Error in a "leave" Statement

Leave f_k. Let us consider the possibilities of error for the atomic action "leave f_k" We are considering an atomic action by itself and not as part of a complex action. The following are possible errors.

(i) missing "leave" action

(ii) leave wrong procedure

(iii) "restart" instead of "leave"

Case(i): Missing leave action. A procedure call always maps into a loop where the loop predicate is given by the equation

$$P_k(I) = ((I=1)\wedge P)\wedge\ ((I>1\wedge R[A_k, f_k]\ (P_k(P, I-1))) \tag{16}$$

Let us assume that the BE for the program with the procedure call is as shown below

$$B^1.\ B^2.\ B^3$$

Where B^1, B^3 are some arbitrary complex BE's and B^2 is the BE corresponding to the procedure call.

The postcondition for the procedure call is given by

$$T[\text{call } f_k](P) = \exists I\ \{L[A_k, f_k](P_k(P,I) \vee T[A_k](P_k(P,I))\} \tag{17}$$

and this will be modified because the L conditions for the procedure are going to be different, since the leave statement is missing.

Since the L condition in the above postcondition is the union of all the "leaves" in the procedure, it is of the form

$$L_1 \cup L_2 \cup \ldots \cup L_n \tag{18}$$

where n is the number of "leaves". Each leave occurs at a different place in the procedure and so the preconditions for each leave can be proven to be distinct; thus the L conditions are distinct, and a missing L condition modifies the L condition shown in equation 18.

The postcondition of the procedure call is the precondition of the

statement following the procedure call. Let the precondition be P_1. This implies

$$P_1 \neq P_1'$$

Thus there will be an error in the predicate of the BE corresponding to the statement following the procedure call. This will show up in B^3.

Case (ii): Leave the wrong procedure. Let us assume that the action is "leave f_n" instead of "leave f_k." In this case the L conditions for both the procedures n and k will be incorrect, as argued in case (i); thus the postconditions for procedures n and k (given by equation 17) will be incorrect.

Since the precondition of the statement following the procedure call is the postcondition of the procedure call, it will also be modified. There the error will show up in the BE for the statement following the procedure call, as an error in the predicate.

Case (iii) Restart instead of leave. The proof is the same as in Case (i) because it appears in the postcondition in the same manner as a missing leave.

E. Error in a "Restart" Statement

"Restart f_k". Possible errors are:

(i) missing "restart"

(ii) "restart" wrong procedure

(iii) "leave" instead of "restart"

Case (i): The R condition for a procedure is the union of all of the "restarts" in the procedure. It is of the form

$$R_1 \cup R_2 \cup R_3 \cup \cdots \cup R_n \tag{19}$$

where n is the number of restarts.

Each restart occurs at a different place in the procedure and so the precondition for each restart can be proven to be distinct; thus the R conditions are distinct, and a missing R condition modifies the R condition in equation 19.

Thus the loop predicate given by 16 is going to be modified. Hence the error manifests itself as an error in the loop predicate of the procedure call.

Case (ii): Restart wrong procedure. In this case the R conditions for two procedures will be incorrect and thus the loop predicates for both of them will be incorrect. Thus the error will show up as an incorrect loop predicate, by the above argument.

Case (iii): Leave instead of restart. Proof is same as Case (i) because it is equivalent to missing "restart".

13.1. Proofs of Complex Actions

A. Error in a Conditional Statement

If b then A_1 else A_2

A_1 and A_2 could be a series of atomic or complex actions. The errors which are possible are

(i) error in b (wrong conditional)

(ii) missing b (missing conditional)

(iii) error in A_1

(iv) error in A_2

The BE for the above complex action is

$$B[A_1](P \wedge b) + B[A_2](P \wedge \sim b) \qquad (20)$$

Case (i): Error in b. This will show up directly in the predicate in the BE of 20.

Case (ii): If the condition is missing the expression in 20 will be modified to either

$$B[A_1](P) \tag{21}$$

or

$$B[A_2](P) \tag{22}$$

Thus the error will show up because the BE's in 21 and 22 cannot be the same as 20.

Case (iii): Error in A_1. The error in A_1 could be either a missing atomic action, an incorrect atomic action, a missing complex action or an incorrect complex action. We have considered the atomic actions and show how errors in them alter the BE's. The proof used here shall be recursive; i.e., assuming that the error in an atomic action will alter the atomic BE, the complex action BE will be altered in the same manner.

B. Error in a Parallel Statement

A_1; A_2

The Behavior Expression for the above complex action is $B[A_1](P)||B[A_2](P)$

Again the error could be in either A_1 or A_2. The argument is the same as for a conditional error.

C. Error in Sequential Statements

A_1 next A_2

The BE for the above action is

$B[A_1](P) \; . \; B[A_2](T[A_1](P))$

The errors which are possible are listed below

(i) error in A_1

(ii) error in A_2

(iii) events out of order i.e., "A_2 next A_1" instead of "A_1 next A_2"

(iv) events not given any order, i.e. A_1; A_2

Case (i): An error in A_1 will show up in $B[A_1]$ using the argument of Case A [Case (iii)].

Case (ii): The proof is the same as in Case A [Case (iii)].

Case(iii): In this case the BE will be modified to

$B[A_2](P) \; . \; B[A_1] \; (T[A_2](P))$

The error will not show up if and only if

$b[A_1](P) \equiv B[A_2](P)$

and

$B[A_2](T[A_1](P)) \equiv B[A_1](T[A_2](P))$

The above conditions will be true if and only if A_1 and A_2 are equivalent pieces of code (i.e., they share the same interaction with the global variables).

Case (iv): Parallel execution instead of serial execution. The BE would indicate this clearly with the "|| instead of "." symbol.

D. Error in a Procedure Call

Call f_k

The possibilities of error are

(i) missing action

(ii) call wrong procedure

Case (i): Missing Action. Each call to a procedure maps into a loop. Consider the BE which has a procedure call

$$B^1 \cdot [B^2]^* I_k \cdot B^3 \tag{23}$$

If the call to the procedure is missing the BE in 23 will be modified to $B^1 \cdot B^2$. Thus the error will show up.

Case (ii): Consider the BE of a program which has a procedure call

$$B^1 \cdot [B^2]^* I_k \cdot B^3$$

Now the wrong procedure is going to be called (i.e., procedure n instead of k). The BE will now be modified to

$$B^1 \cdot [B^{2'}]^* I_n \cdot B^3.$$

These two expressions will be identical if and only if the two procedures are equivalent.