Specification of Interface Behavior for the Automatic Generation of Bus-Interface Models*

Ajay J. Daga, William P. Birmingham

Abstract

This paper describes HIDE, a system that automatically generates VHDL and Verilog bus-interface models (BIMs) from a high-level specification of interface behavior. HIDE users need not be familiar with VHDL or Verilog, and instead specify interface behavior using familiar hardware constructs, such as timing diagrams, state diagrams and truth tables. We present a novel methodology for interface specification that is built on an understanding of the elements of interface behavior. HIDE contains knowledge of the structural elements of an interface specification and the semantics associated with these elements. This allows a user to specify interface behavior in a uniform manner, and automates the task of identifying the relevant semantics associated with a specification. Knowledge of the structure of an interface specification allows formal reasoning on its representation and facilitates the model-generation task.

1. Introduction

Simulation of digital systems is common design practice. For simulation to be effective, accurate models of devices must be available. Complex devices can be described by *behavioral* and *bus-interface models* (BIMs).

Behavioral models capture the full functionality of a device, *i.e.*, both the computation and interface functions are described. For devices such as a microprocessor, these models realize the complete instruction set, allowing small programs to be executed. Behavioral models are difficult to construct, since all details of the component's behavior must be known.

BIMs, also known as hardware-verification models [1], bus-functional models [2], and chip-level models [3] describe only the interface behavior of a device. Essentially, these models capture the communication activity of a device. BIMs treat internal circuitry as a black box, while modeling how a device communicates with its environment. A BIM for a CPU, for example, contains a description of the transactions performed by the CPU (*e.g.*, read, write) to communicate with other devices, but does not model the internal functions of the CPU.

Since BIMs implement a subset of a component's behavior, they are concise, and can be executed quickly on general-purpose hardware. As such, BIMs fit very well with the typical computer-aided engineering (CAE) environment, where a designer captures a design as a schematic and performs logic simulation of that design. BIMs allow a user to ensure that a component communicates correctly with the devices it is connected to for a set of test vectors supplied by the user.

Though less complex than behavioral models, BIMs can be difficult to build because of the complexity of the interface behavior of VLSI devices. Development of simulation models for highly complex VLSI components may require several man-months of effort. This effort is spent in three places: understanding how the device operates from its documentation, understanding the syntactic details of a hardware-description language (HDL), and writing the model. Often, models are

---

* Manuscript received _____. The authors are with the Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, MI, 48109.

This research was supported by Digital Equipment Corporation and the National Science Foundation grant MIPS-905781. All views expressed here are those of the authors, and not necessarily those of the funding agencies.

produced by third parties who are not developers of the component being modeled. For these groups, understanding the operation of a component is a major undertaking; consider that complete descriptions of the operation of complex microprocessors fill several hundred pages. The modeling process is further hampered by HDLs. While HDLs are intended to facilitate model creation, they are sometimes cumbersome and complex, and require considerable programming skill to be effectively used. Thus, a model writer must be an expert in both hardware and a HDL.

To alleviate the problems associated with building BIMs, it is desirable to automate their generation from high-level specifications. Some of the desirable features of automated-model generation are listed below:

- A *means for the specification of interface behavior using constructs that are familiar to a hardware engineer* [4, 5]. This alleviates the need for a hardware designer to be proficient with the syntactic details of an HDL. The inputs to our model-generation process are similar to those used by hardware designers during the synthesis (manual or automated) process to document interface behavior. As a result, engineers can easily generate models for devices they design.

- A *succinct specification methodology* that captures the pertinent information contained in voluminous component manuals, reducing the amount of information needed to generate a BIM.

- A *reduction in the time taken to generate a model.*

- A *method for generating an executable specification of component behavior* (prior to the realization of this behavior in hardware), based on desired signal activity at the interface of a component. This specification may be easily modified, if required, during the design cycle.

To automatically generate BIMs, it is necessary to identify the elements of interface behavior and provide a means for specifying them. The specification methodology should meet the criteria outlined by McFarland [5].

In this paper, we describe a tool, HIDE (HDL Interface Models Designer) [6], that generates simulation models from a graphical specification of interface behavior. The ability to perform this task depends, in large part, on understanding the structure of an interface specification and identifying the semantics associated with this structure. We, therefore, present the rules used to associate formal semantics with structural elements of an interface specification. HIDE may be viewed as a tool that uses embedded knowledge to (a) interpret an interface specification, (b) identify syntactic and semantic errors in the specification, and (c) generate HDL code that encapsulates the behavior contained in the specification.

The key contributions of this paper are listed below:

- A methodology for the automatic generation of BIMs from a high-level specification of interface behavior using timing diagrams, state diagrams and truth tables. There is no existing system that performs this task.

- The identification of semantic rules associated with a graphical specification of interface behavior that permits formal reasoning on an efficient representation of this behavior. McFarland [5] points out that a graphical specification of interface behavior based on timing diagrams is desirable because of its proximity to the way hardware designers represent and communicate hardware information. He, however, faults existing tools that provide a means for the graphical specification of interface behavior with regard to their lack of a rigorous semantics that allows formal reasoning on the specified behavior. The semantic rules that we have outlined in this paper formalize the specification of interface behavior.

The paper is organized as follows. Section 2 defines the elements of interface behavior and discusses our graphical-specification methodology. Section 3 discusses SpecIT, a tool used to capture interface

specifications. The semantic structure of timing diagrams is presented in Section 4. Section 5 defines the elements and important characteristics of an event graph as a representation for timing information. Section 6 presents the event-graph-traversal algorithms. Section 7 discusses the model-generation techniques used to transform the internal representation of an interface specification into simulation models. Section 8 contrasts HIDE with related work and discusses its limitations. Section 9 presents results of the application of SpecIT and HIDE to model generation tasks, and Section 10 summarizes the paper.

2. The Elements of Interface Behavior

The interface behavior of a device may be defined at different levels of abstraction corresponding to the physical, circuit and logical views of a digital device [7]. In this paper, we focus on the specification of interface behavior at the logical level.

Interface behavior may be represented hierarchically through bus cycles, bus states, and signal activity as shown in Figure 2.1.

At the most abstract level, the *bus-cycle level*, the interface behavior of a device is defined by a collection of *bus cycles* that describe how a device communicates with its environment. A bus cycle is a complete interface transaction. A device may execute only one bus cycle at a time. A microprocessor, for example, has read, write, and bus-arbitration cycles.

A bus cycle may be synchronous or asynchronous [8]. A synchronous bus cycle is one where events on an output signal occur at fixed intervals of time relative to the edges of a clock signal. An asynchronous bus cycle does not clock output events at fixed intervals of time. A single interface may have both asynchronous and synchronous bus cycles. The read and write transactions of a CPU may, for instance, be synchronous while bus arbitration may be asynchronous.

The manner in which a bus cycle is initiated results in it being classified as either a *master* or *slave* cycle [8]. A master cycle is initiated by the internal circuitry of a device. For the duration of this cycle, the initiating device is referred to as a *bus master*. A slave cycle is initiated by an external device.

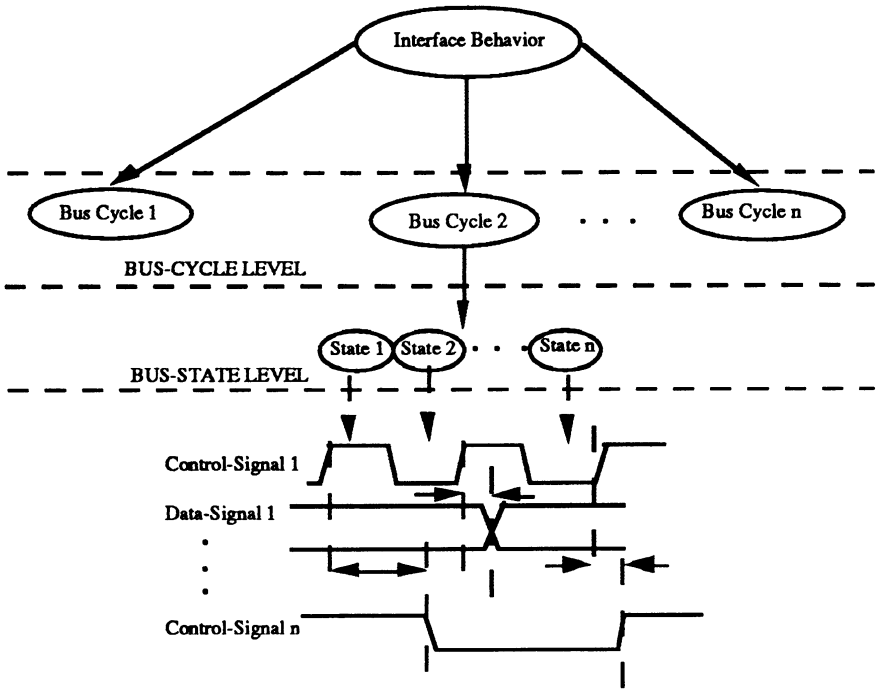

Figure 2.1: Representation of interface behavior.

A bus cycle is composed of a sequence of *bus states*. Sequencing information on the bus states of a cycle are captured at the *bus-state level* of interface behavior. Bus states define sub-activities of a bus cycle, *i.e.*, signal activity on a set of interface ports for a portion of the bus cycle. Bus states represent generic activity and may be shared by different bus cycles. For example, the following sequence of bus states is common to both the read and write cycles of most CPUs:

- place stable address
- assert address strobe
- wait for ready signal
- read or write data
- terminate cycle

A sequence of bus states that form a complete bus cycle is called a *transaction path* for the bus cycle. A read bus cycle for the Motorola MC68020, for example, consists of bus states *S0*, *S1*, *S2*, *S3*, *S4* and *S5* [9]. Taken in that order, these bus states represent a transaction path for the read cycle. A bus cycle may have multiple transaction paths. This, for instance, is the case for the read cycle of the MC68020, as multiple wait states (*S3*) may be inserted during the cycle.

As bus states are shareable among bus cycles, they do not specify exact signal activity. Knowledge of both the bus cycle and the bus state of an interface, however, implies precise signal activity. Figure 2.2 illustrates the notion of generic bus states that have different signal activity associated with different bus cycles. The values taken by a subset of the ports that participate in the read and write cycles of the MC68020, for bus states that are common to both cycles, are shown in Figure 2.2.

CYCLES	SIGNALS	BUS-STATES					
		S0	S1	S2	S3	S4	S5
Write	Read/Write	L	L	L	L	L	L
	Data	Z	Z	Ao	Ao	Ao	Ao
	Data-Strobe	H	H	H	L	L	H
Read	Read/Write	H	H	H	H	H	H
	Data	Z	Z	Z	Z	Ai	Ai
	Data-Strobe	H	L	L	L	L	H

Legend:

L - Logic 0
H - Logic 1
Z - High Impedance
Ao - Active Output
Ai - Active Input

Figure 2.2: Signal activity as a function of bus states and bus cycles.

Signals on an interface are characterized by the kind of information they convey, and may be of type *data* or *control*. Data signals convey information. Control signals communicate the type of bus cycle being executed and control the transfer of information among devices. Some control signals, referred to as *cycle-control* signals, convey information about the *cycle* being executed by a device, for example, the read/write signal on a microprocessor interface. The values of other control signals, however, determine the *bus state* of a bus cycle. These *state-control* signals are used solely to inform other devices of the status of bus-cycle activity. An example of a state-control signal is the data-strobe signal generated by a CPU.

The temporal information associated with signal activity during a bus cycle is typically documented with timing diagrams. These diagrams do not, however, convey other non-temporal information needed for a complete interface specification, such as *conditional information* associated with a bus cycle. Conditional information is required, in the event of multiple transaction paths, to specify transaction paths as a function of signal values. Timing diagrams are also incapable of representing boolean or arithmetic functions that determine signal values or influence other aspects of interface behavior. For instance, the manner in which variable-sized data is multiplexed on a data bus. Thus, in

addition to timing diagrams, state diagrams and truth tables are required to fully specify interface behavior.

2.1. Timing Diagrams

A timing diagram specifies temporal relationships with regard to signal activity within a bus cycle. It consists of a set of signals, and specifies the sequencing of *events* on these signals, for a given bus cycle, through timing links between events. The diagram represents temporal conditions that *must* be satisfied, and documents those that *will* be satisfied in a correctly functioning interface. Timing diagrams focus on events, rather than signal logic levels. An event is a transition between logic levels. An event that occurs on an input signal is referred to as an *input event*, and that on an output signal as an *output event*.

If a bus cycle has a single transaction path, then a timing diagram completely specifies the sequencing of events for a given cycle. This is because a single transaction path implies an identical sequence of events for different executions of the same cycle. In this situation, it is possible to specify temporal relations between relevant event pairs that, taken together, completely specify interface activity.

In the presence of multiple transaction paths, timing diagrams incompletely specify signal activity for a bus cycle. This is because conditional information, used to select among paths, precludes *a priori* knowledge of a transaction path. A transaction path is dependent on the values received by input-control signals. As these values cannot be anticipated, the sequence of bus states that constitute a bus cycle cannot be known prior to the execution of the cycle. Consequently, timing diagrams cannot completely specify the temporal relation between events associated with states that may, or may not, occur. For example, the insertion of wait states during a read or write cycle depends on when a CPU receives a ready signal from the device it is communicating with. As the arrival time of this signal cannot be anticipated, the transaction path in terms of the number of wait states inserted during a bus cycle, cannot be determined *a priori*. Consequently, the temporal relation between a pair of events, where one event occurs before the bus cycle enters a wait state and the other after the bus cycle leaves a wait state, cannot be completely specified on a timing diagram.

A typical timing diagram, for the read cycle of the Motorola MC68020 [9], is shown in Figure 2.3. Address (*ADDR*) is an example of a data signal, while Address-Strobe (*AS*) is a state-control signal. The labels *S0* through *S5* associated with the phases of the *CLK* signal represent different bus states of the read cycle. Note that each of the bus states is associated with events on control signals. During state *S0*, for instance, the *R_W* signal goes high, indicating the start of a read cycle. In state *S1*, signal *AS* goes low indicating valid address on the *ADDR* bus. Not shown on the diagram is how the values of input state-control signals (*DSACK0* and *DSACK1*) determine whether wait states should be inserted. This, however, does not prevent the specification of a timing link, *T14*, indicating the minimum width of the address-strobe pulse. If no wait states are inserted, the address-strobe signal's pulse width will have a duration of at least 120 ns for a 12.5 MHz clock. This is an instance of an incomplete interface specification, as the actual pulse width of the address-strobe signal is dependent on the number of wait states inserted.

2.2. State Diagrams

State diagrams specify sequencing at the bus-cycle and bus-state level (refer Figure 2.1). At the bus-cycle level, a state diagram specifies the initiation and termination conditions of the cycles of a device. Each device has a cycle that captures the default interface activity of a device (the default cycle is referred to as the *Idle* cycle for the rest of this paper). An example state diagram at the bus-cycle level for a memory device is shown in Figure 2.4.

The read and write cycles of a memory device are slave cycles. As a slave cycle is a response to an external bus master, its initiation is dependent on the values of input-control signals (*WE* and *CE* in Figure 2.4). The initiation condition of a master cycle, on the other hand, is controlled by the value

of a *pseudo* signal. A pseudo signal is not present on the interface, but is required to specify the transfer of information between the internal circuitry and the interface. Figure 2.5 shows the bus-cycle-level state diagram for the MC68020 consisting of master read and write cycles[1]. *Next-Cycle* is a pseudo signal that specifies the cycle to be executed at the completion of the current cycle. Note, by default, the *Idle* cycle is executed. Also, note that read and write cycles on the MC68020 may be executed consecutively without intermediate *Idle* cycles.

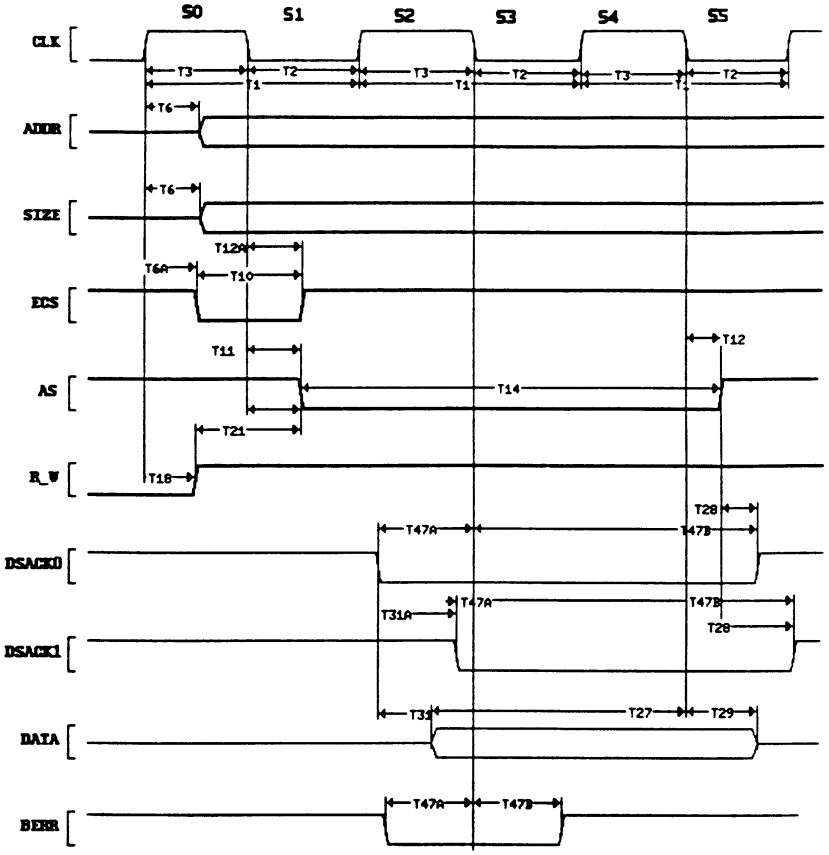

Figure 2.3: An example timing diagram.

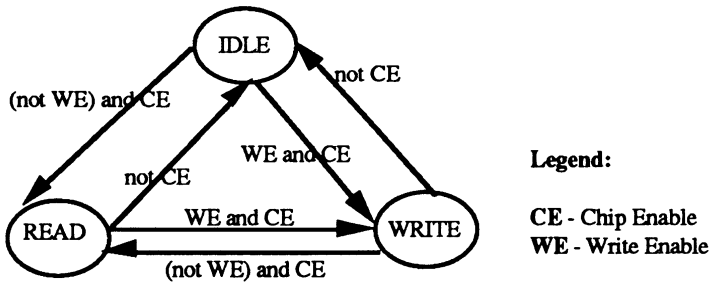

Figure 2.4: Example bus-cycle-level state diagram for slave cycles

[1] The MC68020 has halt, arbitration and read-modify-write cycles; for simplicity only the read and write cycles are shown on the state diagram.

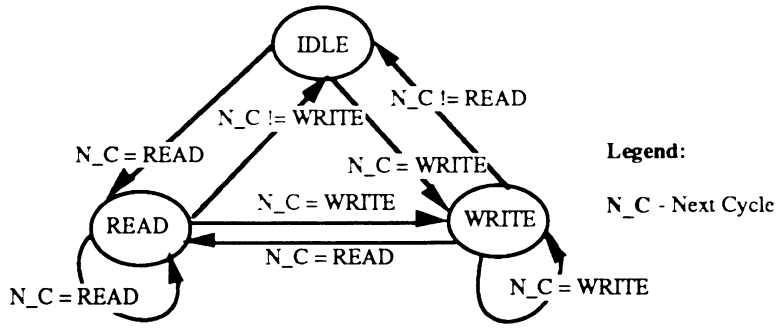

Figure 2.5: Example bus-cycle-level state diagram for master cycles

When a cycle has multiple transaction paths a bus-state-level state diagram is required to specify sequencing and conditional information associated with these paths. We assume that an asynchronous cycle has a single transaction path, and consequently does not need further decomposition at the bus-state level. This is the case for typical asynchronous handshake protocols, such as the fully interlocked handshake [8]. Note that while state diagrams, in general, may also specify signal activity as a function of the state and current input, this information is conveyed through timing diagrams in our methodology.

State diagrams and timing diagrams complement each other, conveying sequencing and signal activity information in a non-redundant manner. In the event of multiple transaction paths, different phases of the signal that sequences an interface through its bus states, referred to as a *sequencing signal* (typically a clock signal), are labeled on a timing diagram with the name of the corresponding state in the state diagram. The timing diagram captures signal activity for each bus state. As the role of a timing diagram is to specify temporal information associated with signal activity, a state diagram does not have to indicate, for example, when a state-control signal should be sampled to determine a state transition. This, in turn, allows an elegant representation of state-sequencing information that is free from temporal concerns.

Multiple bus cycles may be specified through the same state diagram, if they are composed of the same number of bus states and the sequencing between states is the same. This is because state diagrams capture sequencing information without implying specific signal activity. Consequently, two or more bus cycles may share the same bus states and specify different signal activity for each bus state through different timing diagrams. This is the case for most read and write cycles on a microprocessor, for example the MC68020 (shown in Figure 2.2).

Every bus cycle has a single start state and one or more final states. Sequencing between states is dependent on the logic levels sensed at input state-control signals. Conditions on the arcs between bus states specify this sequencing information.

A state diagram specifying the different transaction paths for both the read and write cycles of the MC68020 is shown in Figure 2.6. Arcs are labeled with the transition condition. The sequencing signal is *CLK*. The subscript associated with the sequencing signal represents the event on which the transition is made (*f* for falling edge and *r* for rising edge). *R_P* is a pseudo signal used to indicate the status of a read or write cycle to the internal circuitry.

All read and write cycles have a start state of *S0*, and return to either *S0* (if a bus cycle has to be retried, or if the entire data has not been transferred), S_{halt} (a halt condition), or to the first state of the next cycle (if the current cycle has successfully completed execution). The insertion of multiple *S3* states, depending on the logic level of the *DSACK0* and *DSACK1* signals, represents sequencing information associated with the wait states of a read or write cycle. Note that there is no need to specify when *DSACK0* and *DSACK1* in Figure 2.6 are to be sampled; this information is specified on the timing diagram (see Figure 2.3).

2.3 Truth Tables

Signal values can be arithmetic or Boolean functions of other signals, or functions of their own values during an earlier bus cycle. To specify these functions it is convenient to use a truth table. While timing and state diagrams specify the sequencing of *events* for a bus cycle, a truth table is necessary to specify the manipulation of signal *values*. Truth tables may be used to specify the values taken by both pseudo signals and signals on an interface.

The LHS of a truth table represents *determining signals*; the RHS specifies actions or behavior associated with a set of *dependent signals* that are a function of the determining signals. There are as many truth tables as sets of determining and dependent signals. Each truth table is qualified by the bus cycle(s) to which it applies. A truth table does not have to specify when determining signals are sampled or dependent signals driven.

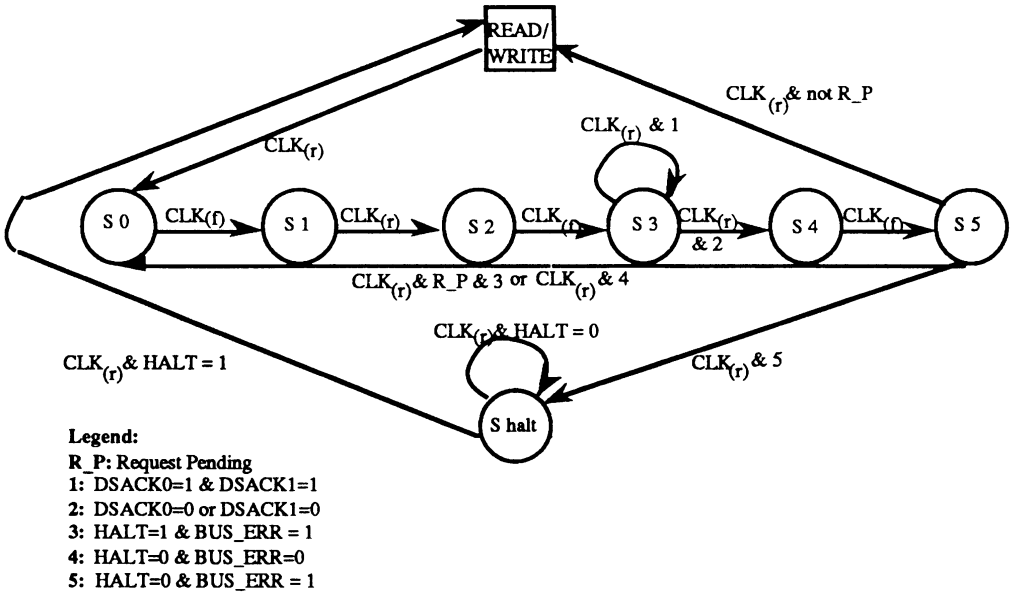

Figure 2.6: State diagram for the read and write bus cycles for the MC68020.

A portion of a truth table representing data multiplexing information for the read and write cycles of the MC68020 is shown in Figure 2.7. The truth table specifies which portion of the data bus is to be read based on address alignment, size of data transfer (word, byte, *etc.*) and size of the device being read. The determining signals are, therefore, *ADDR*[0-1], *SIZE*[0-1] and *DSACK*[0-1][2]. When multiple bus cycles are required to complete an interface transaction, the actions associated with determining the next address and the size of the next transfer also depend on these determining signals. Consequently, the actions associated with determining the values of the *ADDR* and *SIZE* signals for the next bus cycle are listed along with those that specify how to multiplex the data bus.

Datum is a pseudo signal that stores the value of data read. Note how *R_P* is reset to indicate that a read or write cycle has completed its task. Both these signals model the flow of information between the internal circuitry and the interface.

3. SpecIT

Interface specifications using timing and state diagrams and truth tables are captured by the tool **SpecIT** (Specification of Interface Transactions), which generates input to **HIDE**. Simulation models

---

[2] The value received on the DSACK0 and DSACK1 signals on a 68020 determine the data size of the communicating device.

are created by HIDE. The flowchart of the process of generating BIMs from an interface specification is shown in Figure 3.1. SpecIT consists of a timing-diagram editor **Xwave**, a state-diagram editor **Xstate**, and a truth-table editor **Xtable**.

LHS: DETERMINING-SIGNALS						RHS: ACTIONS
SIZE[1]	SIZE[0]	ADDR[1]	ADDR[0]	DSACK1	DSACK0	
0	0	0	0	0	0	DATUM[31:0] = DATA[31:0] R_P = 0
0	0	0	1	0	0	DATUM[23:0] = DATA[23:0] ADDR = ADDR + 3 SIZE = SIZE - 3
0	0	1	0	0	0	DATUM[15:0] = DATA[15:0] ADDR = ADDR + 2 SIZE = SIZE - 2
0	0	1	1	0	0	DATUM[7:0] = DATA[7:0] ADDR = ADDR + 1 SIZE = SIZE - 1
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

Figure 2.7: Example truth table.

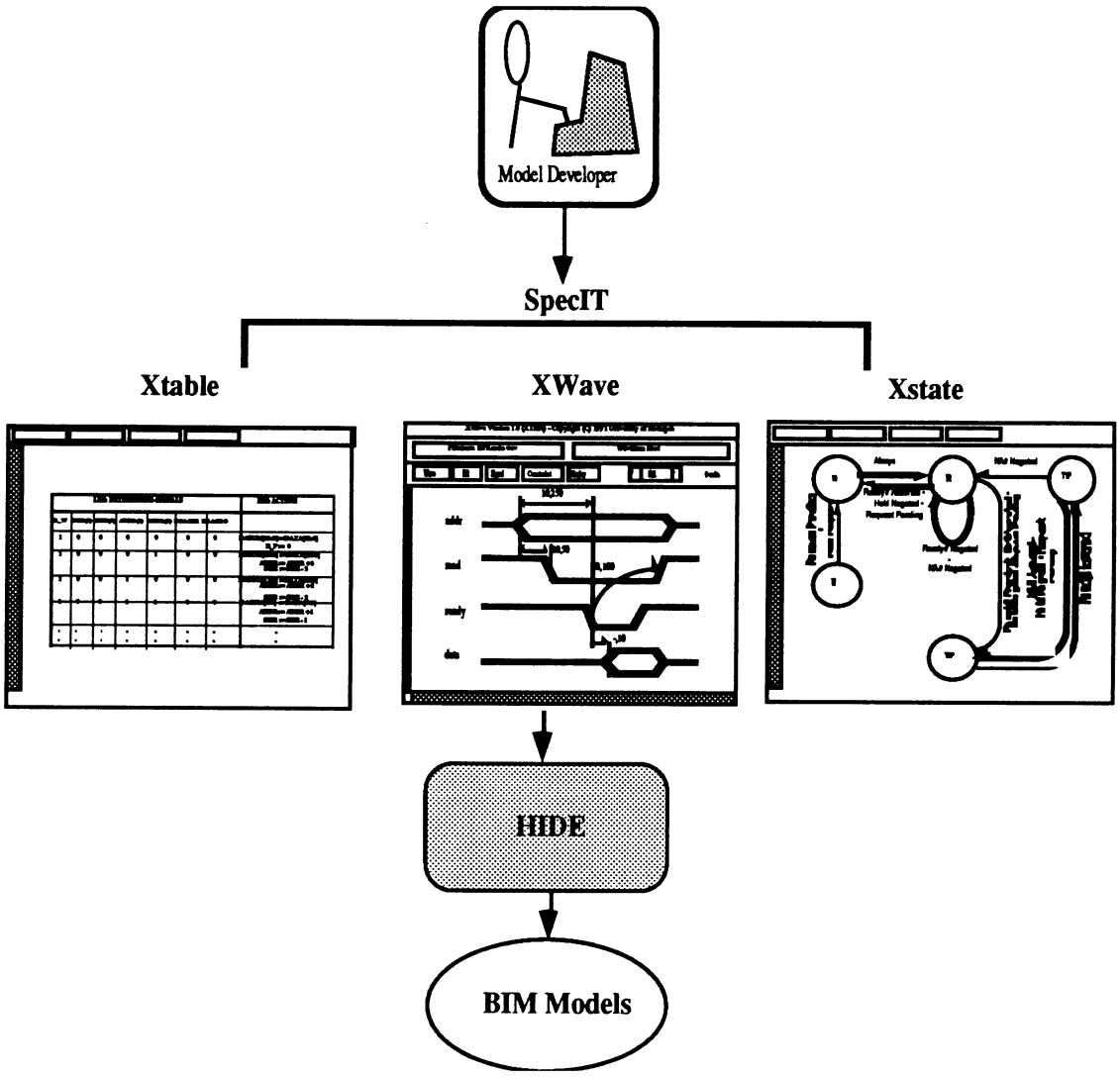

Figure 3.1: Generation of BIMs from an interface specification.

Xwave takes as input a list of the ports, and their characteristics (such as directionality and width), that constitute an interface. It also takes as input a listing of the minimum and maximum values associated with a timing link for all the *operating conditions* of a device. An operating condition for a microprocessor may, for instance, be its clock frequency.

Xwave generates a list of (a) all the events on a timing diagram, and (b) of all the timing links, the pair of events connected by them and the link type (causal or constraint). Figure 2.3 illustrates a typical timing diagram entered through Xwave. Xwave uses timing-diagram conventions outlined by Rony [10].

Xstate generates a state-transition table A typical state diagram captured by Xstate is shown in Figure 2.6.

Xtable allows the user to enter a truth table. Signal values on the LHS of the truth table may be 0, 1 or X. The RHS of the table associates each row with a set of actions. Figure 2.7 illustrates an example truth table entered using Xtable.

4. Timing-Diagram Semantics

Timing-diagram semantics are culled from the structure of timing links and the events they connect. Identifying the semantics associated with timing links is key to creating an appropriate representation for timing information, and greatly facilitates the model-generation task. In this section, we discuss the types of timing links, the timing link structure that indicates signal sampling, and contrast our method for culling semantics from an interface specification with other work in this area.

4.1 Causal and Constraint Timing Links

It is important to differentiate between causal and constraint links when generating simulation models. A causal link results in code that drives signal values after a specified duration, while a constraint link requires a check to determine if the time between events is within limits. It is desirable to allow a user to enter timing links without having to specify their type.

A timing link t_l, as shown in Figure 4.1, connects a pair of events e_{from} and e_{to}. The event occurring earlier relative to t_l is e_{from}, and the later one is e_{to}. Timing links may be of only two types, causal or constraint. A link can either *specify* or *constrain* the time at which an event will occur relative to another event. A link that specifies a time of occurrence for an event e_{to} relative to an event e_{from} is *causal*. A link that constrains the time of occurrence of an event e_{to} relative to an event e_{from} is *constraint*. This notion of causal and constraint links is similar to that outlined by Martello, *et al.* [11].

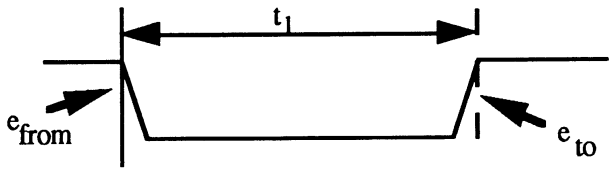

Figure 4.1: A timing link.

The timing-link type may be determined by the pair of events it connects. The rules for identifying link type are given below.

Rule 4.1:

A timing link is of type *constraint* if e_{to} is an input event. Links 1 and 4, in Figure 4.2, are of type constraint.

By definition, a device can never *cause* an event to happen on its input signal; it can only monitor input signal activity. Consequently, a timing link that connects an event (on an input or output signal) e_{from} with an input event e_{to} has to specify a *constraint* on the temporal relation between these two events. For example, in Figure 2.3, link *T28*, with e_{from} being the rising edge of *AS* (an output event) and e_{to} being the rising edge of *DSACK0* (an input event), is of type constraint. The rising edge of *DSACK0* is caused by the device with which the MC68020 is communicating. Hence the rising edge of *AS* cannot *cause DSACK0* to rise.

Rule 4.2:

> A timing link is *causal* if e_{to} is an output event. The only exception to this rule is indicated in Rule 4.4. Links 2 and 3 in Figure 4.2 are causal.

A device *causes* events to happen on its output signals. The temporal separation between an event e_{to} on an output signal and an event e_{from} (on an input or output signal) is a *known* value (dependent on the delay of the circuitry that drives the output signal) and is specified on a timing diagram. It is, therefore, incorrect for a link to specify a constraint over a period of separation that is known and invariant.

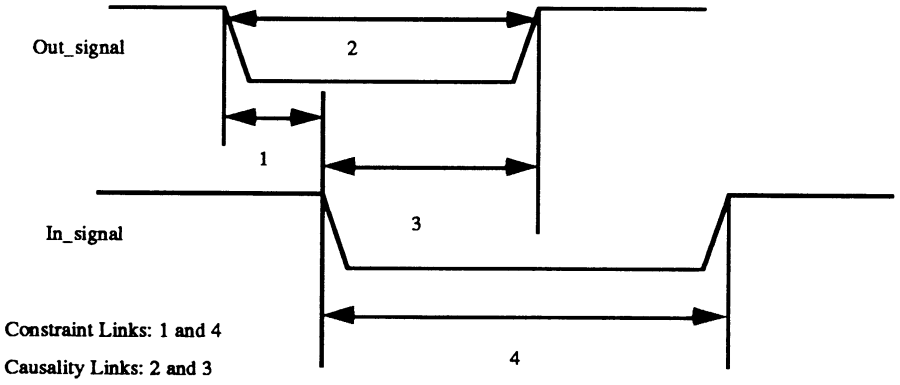

Figure 4.2: Causal and constraint timing-link types.

Causality, in the sense that it is used here, does not necessarily imply that e_{from} causes e_{to}. Event e_{to} may have many causal links associated with it, and the task of determining the **true** causal link is handled by the event-graph-traversal algorithm (see Section 6.2). For example, consider the timing diagram in Figure 2.3, and let e_{to} be the rising edge of signal *AS*. Links *T14* (e_{from} is the falling edge of *AS*) and *T12* (e_{from} is the falling edge of *CLK* in state *S4*) are both causal, as e_{to} is an output event. Link *T12* is the *true* causal link; link *T14* is shown for documentation purposes.

4.2 Sampling Structure

The sampling structure on a timing diagram implicitly specifies when a signal value should be captured. It also alleviates the need for state diagrams and truth tables to specify temporal information regarding the sampling of signal values referenced by them.

A synchronous cycle samples input signals relative to events on another signal. Typically, signals are sampled on the falling or rising edges of a clock signal, as is the case with the *CLK* signal in Figure 2.3[3].

An asynchronous cycle does not share the same notion of signal sampling. Data signals are sampled relative to events on one or more control signals. Consider, for instance, the timing diagram for a memory interface shown in Figure 4.3. The address and data values for a write operation are sampled at the rising edge of either the chip-enable (*CE*) or write-enable (*WE*) signals.

The sampling of a signal is shown implicitly on a timing diagram by the structure of timing links shown in Figure 4.4. All sampled signals must meet set-up and hold-time constraints relative to the sampling signal. Link t_S in Figure 4.4 is the set-up constraint, and link t_h is the hold constraint. For generality, signals are shown using the active logic level, but the structure is valid for all logic levels.

Rule 4.3:

Any input signal on a timing diagram whose two consecutive events e_1 and e_2 have timing links to a common event e_S on an *input* signal has its logic level l_S, between e_1 and e_2, sampled at e_S.

The justification of this rule follows from the earlier discussion on set-up and hold time requirements that an input signal has to satisfy. These requirements are structurally manifested as shown in Figure 4.4.

Rule 4.4:

If the sampling event e_S occurs on an output signal, then the link between e_1 and e_S is of type constraint and not causal as specified by Rule 4.2.

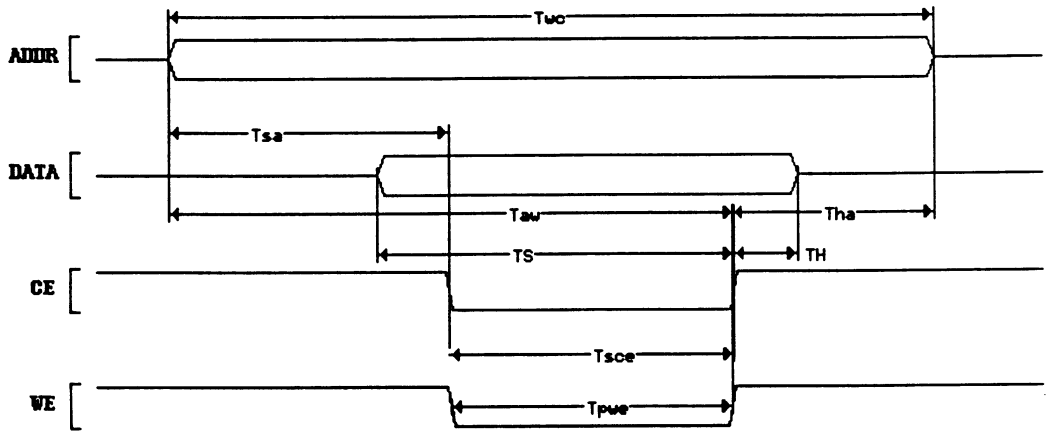

Figure 4.3: Timing diagram for a write cycle on a memory device.

This is the only exception to Rule 4.2, wherein events on an output signal are used by a device to sample input signals. In this situation, the event e_1 does not cause e_S, and the link between these events specifies the set-up time constraint. In Figure 4.4, if *sampling-signal* were an output signal and *sampled-signal* an input signal, then link t_S would be of type constraint, and not causal as indicated by Rule 4.2.

---

[3] Some devices sample signals at high or low levels of a clock signal. This does not change the following discussion on the sampling structure of timing links.

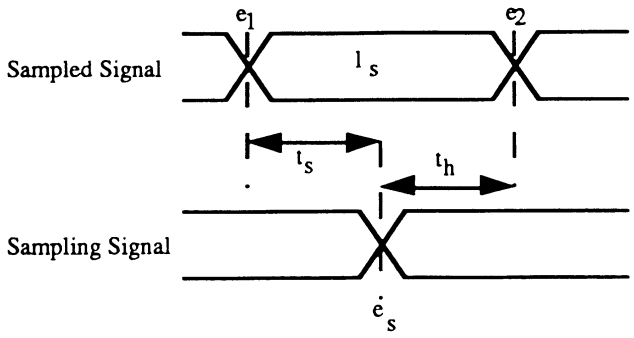

Figure 4.4: Sampling structure.

An asynchronous interface, unlike a synchronous interface, may show signals sampled by more than one state-control signal. Consider, for example, the sampling of the signal *DATA* in Figure 4.3. *DATA* has t_S and t_h links with regard to the rising edge of both *CE* and *WE*.

Rule 4.5:

In the case where a logic level l_S is sampled by a multiple event set E = $\{e_{S1}, e_{S2}, ..e_{Sn}\}$, the sampling of l_S is performed at the event $e_{Si} \in$ E that occurs before the other events $e_{Sj} \in$ E.

For example, in Figure 4.3 if the rising edge of *CE* (*WE*) were to occur before *WE* (*CE*), then the value of *DATA* when *CE* (*WE*) rises is sampled by the device. Note that this rule, in effect, assumes that the rising edges of *CE* and *WE* are logically ORed to sample *DATA*. This rule may be overridden by annotating the constraint links leading from the rising edges of *CE* and *WE* to *DATA* by any other logical relation (*e.g.*, AND).

4.3 Discussion

Borriello [7, 12] identifies three kinds of constraints representing *simultaneity, ordering* and *synchronicity* constraints; causal links are not discussed. We have shown that only causal and constraint links are necessary for timing diagrams. A constraint link, based on the minimum and maximum values associated with it (whether these values are positive, negative, zero or infinity), *implicitly* represents Borriello's three different constraint types. Borriello does not identify rules to cull sampling and link-type semantics. A signal that is synchronous to a clock is explicitly labeled so by a user. Waves restricts the drawing of synchronous signal transitions based on set-up and hold timing constraints relative to the sampling edge of a clock signal.

The specification mechanism developed by Subrahmanyam [13, 14] associates semantics with graphical icons. A user selects the appropriate icon to explicitly specify the desired semantics. The ability to identify semantics from structural elements is not embedded in the tool.

Mavaddat and Gahlinger's [15] definition of a constraint link (referred to as a *required constraint*) as being a bound on a pair of input events is *overly restrictive*, as a constraint may be specified between an output and input event when the output event is e_{from} (Rule 4.1). On the other hand, their definition of a causal link (referred to as a *produced constraint*) as being a bound on the separation between a pair of events when one of these is an output event, is *overly accommodative*; Rule 4.2 states that a causal link has to be one where the e_{to} event is an output event.

5. Event Graphs

A natural representation of temporal information is an *event graph* [16]. An event graph is a directed acyclic graph (DAG) whose nodes, E = $\{e_1, e_2...,e_n\}$, represent events, and arcs, A = $\{t_1, t_2...,t_m\}$, represent *causal* links between pairs of events. Constraint links are not shown on the event graph. An

event graph is essentially a timing diagram, without constraint links, where logic levels have been abstracted away. The graph is acyclic as an event, e_i, occurring after an event, e_j, cannot, in turn, cause e_j to occur. A causal link, t_l, connecting e_{from} with e_{to} is represented by a directed arc from e_{from} to e_{to}. Each causal link has a minimum value, $min(t_l)$ and a maximum value, $max(t_l)$. The minimum separation between a pair of events may be 0 units, while the maximum may be ∞ units. The event graph for the timing diagram of Figure 2.3 is shown in Figure 5.1.

Traversing an event graph refers to the propagation of timing values from a set of reference events $\mathbf{R} = \{e_{r1}, e_{r2}...,e_{rk}\}$ *to events that are connected directly or indirectly to R.* Event-graph traversal is necessary to: (a) identify a subset of the events on a timing diagram that serve as temporal reference points for other events, and (b) define the temporal separation of a non-reference event, connected indirectly to a reference event. through a direct link that subsumes the information contained in the indirect links.

An event e_i is connected *directly* to an event e_j when the length of the path connecting these events is exactly one. An event e_i is *indirectly* connected to an event e_j when the length of the path connecting these events is greater than one. An event graph's *depth* is the length of the longest path from a reference event e_r to a leaf event e_i that has an out-degree of zero. After the traversal of an event graph its depth is exactly one.

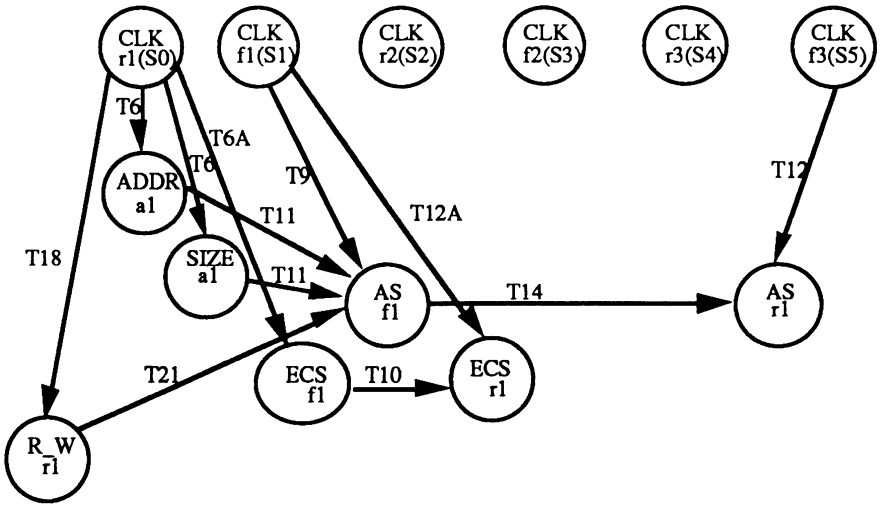

Figure 5.1: Example event graph.

Traversal occurs by following arcs leading from a reference event e_r to all non-reference events $e_i \in$ E rooted at $e_r \in$ R. A reference event e_r has an in-degree of zero and out-degree of at least one. An event graph may have more than one reference event. Every graph has a start event e_s that is a reference event and is the first event to occur in a bus cycle. The event e_s is specified by a user. The notion of a reference event is similar to the *anchor set* described by Martello, *et al.* [16], Ku and DeMicheli [17] and the *reference frame* outlined by Wallace and Sequin [18].

The propagation of timing values refers to determining the *range of occurrence* associated with each event e_i, relative to an event e_r. This range of occurrence, $t_p(e_i)$, has the form $t_{min}(e_i) \leq t_p(e_i) \leq t_{max}(e_i)$. The time $t_{min}(e_i)$ ($t_{max}(e_i)$) depicts the earliest (latest) that event e_i could occur relative to e_r, and is computed using the $min(t_l)$ ($max(t_l)$) values of timing links connecting e_r with e_i. The time $t_{min}(e_i)$ ($t_{max}(e_i)$) is a lower bound (upper bound) on the temporal separation between e_r and e_i. The notion of a range of occurrence is similar to that of a *temporal interval* outlined by Allen [19].

Every event e_i has an initial range of occurrence $0 \leq t_p(e_i) \leq \infty$ relative to e_s. The event e_s has $t_{max}(e_s)$ and $t_{min}(e_s)$ equal to 0. The objective of traversing an event graph is to restrict the range of

occurrence of all output events relative to a reference event. An event e_i could have multiple reference events e_r, in which case a set of rules is required to determine the true reference event. Listed below are rules that characterize the range of occurrence associated with an event.

Rule 5.1:

The range of occurrence of an input event e_i cannot be further restricted from the initial value of $0 \leq t_p(e_i) \leq \infty$ relative to the start event e_s.

There is no causal link whose e_{to} event is an input event. As an event graph is composed of only causal links, no input event may have an arc leading in to it. Consequently, an input event is not traversed during the propagation of timing values, and retains its initial range of occurrence.

Rule 5.2:

After traversing an event graph, all output events e_i have $t_{max}(e_i)$ less than ∞. A value of ∞ for $t_{max}(e_i)$ manifests an incorrect interface specification.

If an output event has an unbounded range of occurrence (*i.e.*, ∞), then no other device would be able to communicate with it. Every output event must have a finite range of occurrence relative to some reference event.

Rule 5.3:

The range of occurrence for a reference events e_r is $0 \leq t_p(e_i) \leq \infty$ relative to the start event e_s, unless e_r is e_s, in which case $t_{max}(e_r)$ and $t_{min}(e_r)$ have values of zero. The exception to this rule is discussed in Rule 6.2.

As a reference event e_r has an in-degree of zero, its range of occurrence cannot be changed during the traversal of an event graph.

Rule 5.4:

No output event may be a reference event unless it is the start event e_s or a sequencing event on an output signal. All reference events are, therefore, input events, with the exception of e_s and those events that conform to Rule 5.6. Not all input events are reference events, as some input events may have an out-degree of zero.

Using Rule 5.4, an output event, unless it is the start event always has an arc incident on it. Thus, it may not retain its default value of ∞ for $t_{max}(e_i)$. Therefore, by definition, an output event cannot be a reference event with the exception of e_s and events that satisfy Rule 5.6.

Events on a sequencing signal are called *sequencing events*, and each sequencing event e_i is associated with a bus state s_i, the state entered by an interface when the event e_i occurs.

Rule 5.5:

The set of events **E** rooted at a sequencing event e_i have their causality superceded by the bus state of an interface. The events in **E** occur when the device enters the bus state s_i associated with e_i.

The rationale for this rule is that the event at which a device enters a state s_i may or may not be the particular sequencing event e_i shown on a timing diagram. For example, in the event graph shown in Figure 5.1, CLK_{f3} has a causal link to AS_{r1}. If there are no wait states inserted during a read cycle, then CLK_{f3} would cause AS_{r1}. If, however, one wait state is inserted, then CLK_{f4} would cause AS_{r1}. In general, $CLK_{f(3+n)}$ is the event that causes AS_{f1} (n is the number of wait states inserted). In all cases, however, AS_{r1} occurs when the read cycle enters state *S5*.

Rule 5.6:

All sequencing events that have an out-degree of at least one are reference events.

This rule follows from Rule 5.5. As all events E rooted at a sequencing event e_i are caused when the cycle enters state s_i associated with e_i, their range of occurrence is defined relative to s_i. The reference event for events in the set E is therefore e_i.

6. Event-Graph Traversal

HIDE uses a list of events, provided by Xwave, to generate an unconnected event graph. HIDE then connects events that are causally linked. As constraint links are not part of the event graph, we will delay a discussion on how they are processed until the section on code generation. HIDE traverses an event graph using the **traverse_graph** algorithm, which calls **propagate_value** to propagate timing values for a reference event. The propagate_value algorithm calls **resolve_reference_event** to resolve the situation where an event has multiple reference events. These algorithms are discussed in the following sections.

6.1 The Resolve_Reference_Event Algorithm

Consider the case, depicted in Figure 6.1, where an event e_i has two reference events e_{r1} and e_{r2}, and that e_i has received a range of occurrence relative to e_{r1}. When the event graph relative to e_{r2} is traversed e_i will be reached. At this point a decision has to be made, if possible, on which of the two reference events, e_{r1} or e_{r2} to use when determining e_i's range of occurrence. The resolve_reference_event algorithm performs this task, and uses the following rule.

Rule 6.1:

If an event e_i (an e_{to} event) is indirectly or directly connected to events e_{r1} and e_{r2} (e_{from} events), and if it is known that e_{r1} occurs before e_{r2}, then the link between e_{r2} and e_i is the true causal link. The link between e_{r1} and e_i is shown for documentation purposes.

The justification of this rule is as follows. Given that e_{r1} occurs before e_{r2}; assume, contrary to Rule 6.1, that the link t_{r1}between e_{r1} and e_i is the true causal link and that between e_{r2} and e_i is a link, t_{r2}, shown for documentation purposes. The event e_i has to occur after e_{r2} for t_{r2} to exist, and therefore the chronological sequence of these events *must* be e_{r1}, e_{r2}, e_i. While e_{r2} may be *constrained* to occur within a temporal span relative to e_{r1}, *the actual time at which* e_{r2} *occurs is not known or specifiable.* The event e_{r2} may never occur during an erroneous execution of a bus cycle. It is therefore incorrect to *document* the separation between e_{r2} and e_i. Consequently the assumption that t_{r1} is the true causal link is incorrect. On the other hand, *as* e_{r1} *has to have occurred for* e_{r2} *to occur,* it is always correct to document the separation between e_{r1} and e_i.

Rule 6.1 is shown in Figure 6.1. The event CLK_{f1} (e_{r2}) is the true reference event in this situation. The application of this rule in different cases is listed below.

Case A:

If e_{r1} (e_{r2}) is e_s, then the range of occurrence is taken relative to e_{r2} (e_{r1}). This is because no event can occur before e_s.

Case B:

If e_{r1} and e_{r2} are non-sequencing events on the same signal, and it is known that e_{r1} (e_{r2}) occurs before e_{r2} (e_{r1}), then e_{r2} (e_{r1}) is the true reference event. The ordering of events on a non-sequencing signal is explicitly shown on a timing diagram.

Case C:

If e_{r1} and e_{r2} are sequencing events, then event e_{r1} is associated with bus state *s1* and e_{r2} with bus state *s2*. If it can be shown, by traversing the state diagram, that state *s1* (*s2*) always occurs before state *s2* (*s1*), then e_{r2} (e_{r1}) is the reference for event e_i. The traversal of a state diagram to determine if state *s2* occurs after state *s1* requires two conditions to be satisfied.

1. It must be shown that state *s1* **has** a transition path to state *s2*, and

2. that state *s2* **does not** have a transition path to state *s1*.

If condition 1 is not satisfied, then nothing can be said about the temporal relation between the two states *s1* and *s2*. If condition 2 is not satisfied, then *s1* may or may not occur after *s2*, and consequently no *a priori* decision can be made on the temporal relation between the two states. If both conditions are not satisfied both e_{r1} and e_{r2} are retained as reference events.

The search for a transition path between two states is performed by a depth-first search of the state diagram. The start state for condition 1 is *s1*, and that for condition 2 is state *s2*. A state once visited is marked, and is not revisited. Arcs leading into the start state (*S0* in Figure 2.6) are not traversed as they represent the start of a new bus cycle. The search terminates for condition 1 (2) when state *s2* (*s1*) is reached, or when there are no more arcs to traverse.

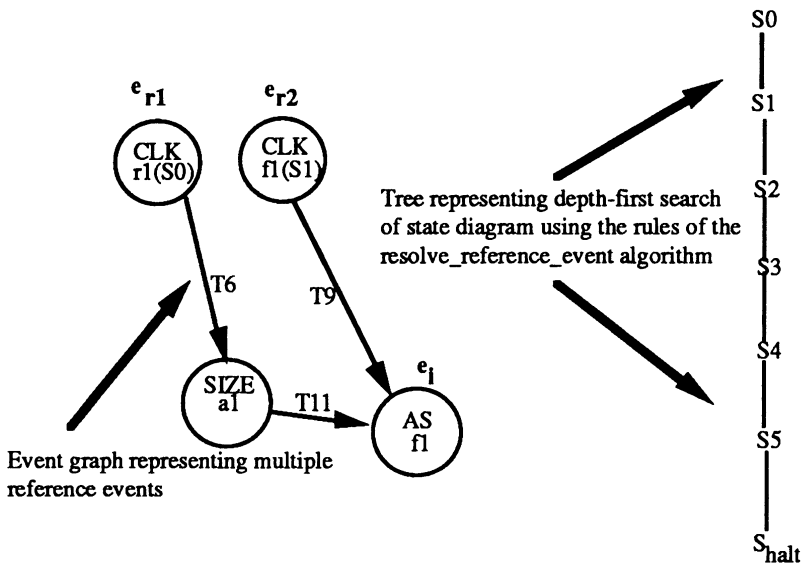

Figure 6.1: Illustration of Case C in resolve_reference_event.

A portion of the event graph in Figure 5.1 that corresponds to case C is shown in Figure 6.1. The tree representing the depth-first search of the transition diagram is also shown in Figure 6.1. From the tree shown in Figure 6.1, it is clear that *S0* has a path to *S1*, and that *S1* does not have a path to *S0*. Therefore, bus state *S1* is always reached after *S0* and, consequently, the falling edge of the clock signal is the true reference event for the falling edge of *AS*.

Case D:

If events e_{r1} and e_{r2} do not conform to scenarios A, B or C, then e_i's range of occurrence relative to both e_{r1} and e_{r2} is maintained.

6.2 The Propagate_Value Algorithm

The propagate_value algorithm is given a reference event e_r, and traverses the event graph affixing a range of occurrence for events rooted at e_r. An event e_i has a set of parent events **P** = $\{e_{p1}, e_{p2}...,e_{pm}\}$ and children events C = $\{e_{c1}, e_{c2}...,e_{ck}\}$. The algorithm performs a breadth-first search of the event graph rooted at e_r by traversing arcs connecting a parent event with its children. All the arcs connected directly or indirectly to e_r are traversed, except as indicated by Rule 6.2.

Rule 6.2:

If event e_r is a sequencing event and is associated with bus state s_r, and is connected directly or indirectly to a sequencing event e_i (associated with bus state s_i) that occurs on the same signal, then the arcs **out** of e_i are not traversed during a traversal of the event graph rooted at e_r. This is illustrated in Figure 6.2. The set of links $\mathbf{E_2}$, shown dashed, are not traversed. Note the following:

- $e_r \neq e_i$ and $s_r \neq s_i$.
- e_i **is** traversed during a traversal of e_r.
- e_i is an output event, as only then would it have a causal link incident on it (Rule 4.2). Consequently, the sequencing signal in this situation has to be an output signal.

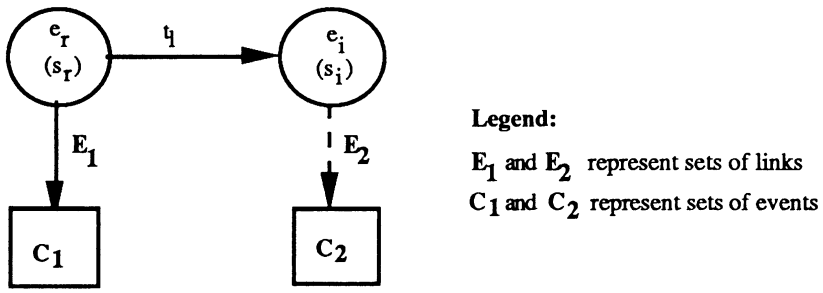

Figure 6.2: Illustration of Rule 6.2.

This rule on event-graph traversal follows from Rules 5.5 and 5.6. As all sequencing events that have an out-degree of atleast one are reference events, the graph rooted at e_i cannot be traversed when the graph rooted at e_r is being traversed. If traversed, events caused by s_i would have their range of occurrence defined relative to s_r. For example, assume that the *CLK* signal in Figure 2.3 is an output signal (it is actually an input signal). If links *T1*, *T3* and *T12* are followed from CLK_{r1} then the falling edge of the *AS* signal would have a range of occurrence defined by the equation $2T1+T3+T12$, relative to CLK_{f1}. This is incorrect as it implies that the *AS* signal is raised, and the read cycle terminated, regardless of the number of wait states inserted during a read cycle.

The impact of traversing arcs leading into reference events on an output sequencing signal, but not the arcs leading out of these events, is the following:

- Those reference events that occur on an output sequencing signal have an in-degree greater than zero.

- Also, reference events on an output sequencing signal have a range of occurrence that is not $0 \leq t_p(e_i) \leq \infty$ relative to the start event e_s. This is the exception to Rule 5.3.

A breadth-first search of the event graph is necessary for the following reason. Consider an event e_i with children C and parents e_{p1} and e_{p2}, as shown in Figure 6.3. C represents the event graph rooted at e_i, and could have a depth greater than one. A depth-first search of the event graph would result in the children C receiving their values based on the value received by e_i ($25 \leq t_p(e_i) \leq 90$) through e_{p1} (assuming that e_{p1} is traversed first). If, however, when the link between e_{p2} and e_i is traversed, it is determined (using rules for affixing a range of occurrence) that e_i's value ($20 \leq t_p(e_i) \leq 80$) is dependent on that of e_{p2}, then the arcs between e_i and its children must be revisited. This is wasteful. A breadth-first search never requires an arc to be revisited. The rules for affixing a range of occurrence for an event follow.

Rule 6.3:

If e_p is e_r, then $t_p(e_C)$ for a child e_C that **has not** been visited, is defined so that the lower (upper) bound of $t_p(e_C)$ inherits the $min(t_l)$ ($max(t_l)$) value of the arc t_l connecting e_p with e_C.

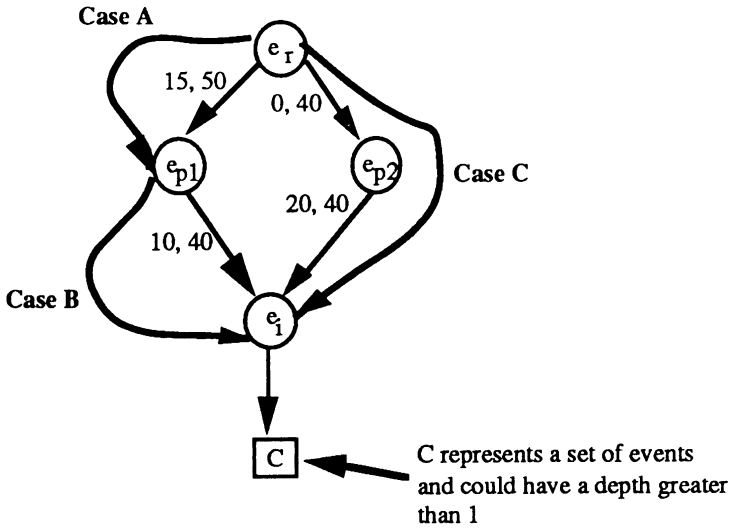

Figure 6.3: Need for a breadth-first search of the event graph.

The justification of this rule is straightforward; the range of occurrence of a node directly connected to a reference event is dependent on the value of the link connecting the two events. This rule is illustrated by Case A in Figure 6.3. When the link between e_r and e_{p1} is traversed, e_{p1} has not been previously visited. It receives a value of $15 \leq t_p(e_{p1}) \leq 50$.

Rule 6.4:

If e_p is not e_r, then e_p will have been visited. If e_p is connected through arc t_l to a child e_C that **has not** been visited, then:

- $t_{min}(e_C) = t_{min}(e_p) + min(t_l)$
- $t_{max}(e_C) = t_{max}(e_p) + max(t_l)$

  A check is made to ensure that $t_{min}(e_C) \leq t_{max}(e_C)$.

The range of occurrence of a node that is connected indirectly to a reference event is the sum of the $max(t_l)$ ($min(t_l)$) values on the path connecting the pair of nodes. This is shown labeled Case B in Figure 6.3. When the link between e_{p1} and e_i is traversed (assuming e_{p1} is traversed before e_{p2}), e_i

has not yet been visited. The range of occurrence of e_i, $t_p(e_i)$, receives a value that is dependent on the link t_l being traversed and $t_p(e_{p1})$. The range of occurrence $t_p(e_i)$ is therefore, $25 \leq t_p(e_i) \leq 90$.

Rule 6.5:

If e_C **has** been visited and has a range of occurrence relative to e_r then:

- $t_{min\text{-}new} = t_{min}(e_p) + min(t_l)$
- $t_{max\text{-}new} = t_{max}(e_p) + max(t_l)$
- $t_{min}(e_C) = max\ (t_{min\text{-}new}, t_{min}(e_C))$
- $t_{max}(e_C) = min(t_{max\text{-}new}, t_{max}(e_C))$
  As in Rule 6.4 a check is made to ensure that $t_{min}(e_C) \leq t_{max}(e_C)$.

Event e_C's range of occurrence is restricted, so that the $t_{min}(e_C)$ ($t_{max}(e_C)$) bound satisfies both the existing specification on the temporal separation between e_r and e_C, and the $t_{min\text{-}new}$ and $t_{max\text{-}new}$ values. The rationale behind the rule is the following. In the event of multiple direct or indirect arcs from e_r to e_C, the lower (upper) bound should be established such that all the specifications on the minimum (maximum) separation between e_r and e_C are simultaneously satisfied. The maximum (minimum) value of a set of lower (upper) bound specifications satisfies all other lower (upper) bound specifications.

Rule 6.5 is illustrated in Figure 6.3, labeled Case C. The lower bound of $t_p(e_i)$ is 25 when the link connecting e_{p2} with e_i is traversed. As $t_{min\text{-}new}$ is equal to 20 and is less than $t_{min}(e_C)$ the lower bound of $t_p(e_i)$ remains unchanged. The value of $t_{max}(e_i)$ is 90 and that of $t_{max\text{-}new}$ 80. Consequently, $t_{max}(e_i)$ is set to 80 and the range of occurrence of e_i is revised to $25 \leq t_p(e_i) \leq 80$.

Finally, if e_C has a range of occurrence relative to an event $e_{r1} \neq e_r$, then the resolve_reference_event algorithm is called to determine the true reference event.

The result of the propagate_value algorithm for the event graph shown in Figure 5.1 is shown in Figure 6.4. The arc connecting a reference event, e_r, with an event e_i is labeled by the t_{min} and t_{max} values of e_i's range of occurrence relative to e_r.

6.3 The Traverse_Graph Algorithm

The traverse_graph algorithm passes events to the propagate_value algorithm that have no parents or occur on an output sequencing signal, and thus are reference events (Rule 5.4). The algorithm starts by passing the start event e_s to propagate_value. It then looks for arcs not yet traversed, and if any such arc exists, it passes a reference event e_r to the propagate_value algorithm. The traverse_graph algorithm terminates when all arcs have been traversed. Figure 6.4 shows the result of the traverse_graph algorithm, as all the reference events associated with the event graph in Figure 5.1 are shown.

Rule 6.6:

If an arc t_l remains to be traversed and the event e_{from} associated with t_l is an output event, then the event e_r passed to the propagate_value algorithm is the first input or sequencing event found during a search up the event graph that is connected to e_{from}.

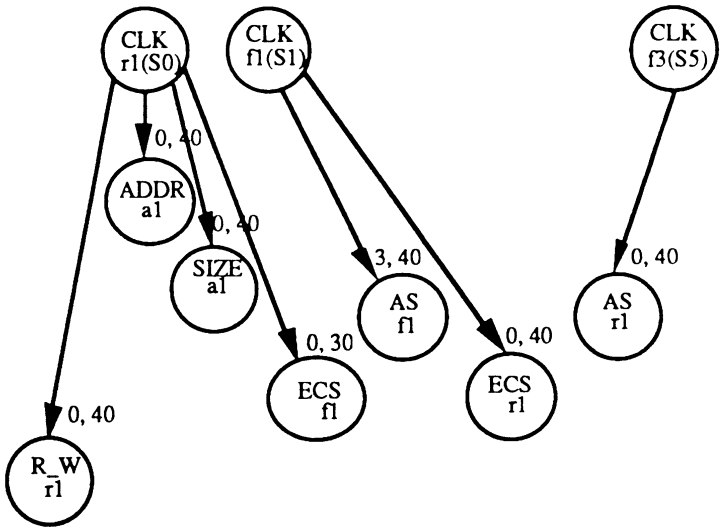

Figure 6.4: Result of the propagate_value algorithm for the entire event graph.

Note that in the absence of an output sequencing signal there has to be an input event connected directly or indirectly to e_{from}, as the only output event that can be a reference event in this situation is e_S. The first input event found on this recursive search is e_r, as it cannot have any parents (if it did it would be an input e_{to} event for a causal link, violating Rule 4.2). Also note that none of the events on this search up the event graph will have a range of occurrence associated with them, except if they are sequencing events. This is because if they did, then link t_l would have been traversed.

Rule 6.7:

If an arc t_l remains to be traversed, and the event e_{from} associated with t_l is an input event, then the event e_r passed to the propagate_value algorithm is e_{from}.

As there can be no input event on an event graph with an arc leading in, the first input event found (e_{from}) when searching up the event graph from e_{to} must be a reference event.

6.4 Discussion

Mavaddat and Gahlinger [15] use an event graph to represent timing information and then *tighten* (compute the shortest paths between events) this event graph. Tightening is similar to our traversal of the event graph. Prior to tightening, their event graph may have cycles as a maximum constraint between a pair of events is represented as a negative weighted edge with a direction opposite that of a minimum constraint. Ku and DeMicheli [17] and Khordoc *et al.* [20, 21] use a similar representation. We, however, consider the entity of relevance to be the *range of occurrence* of an event. Separating the representation of lower and upper bounds does not allow the manipulation of a range of occurrence. The rules that define ranges of occurrence help in identifying errors; for example, the detection of an output event with an infinite upper bound. The only error detectable in the scheme used by Khordoc *et al.* is a lower bound exceeding the upper bound (manifest as a cycle with positive weight).

Martello and Levitan [16, 22] use an event graph, for interface verification tasks, that is similar to ours. Each event in their graph has an enabling expression that contains conditional information that may be optionally specified to affect the occurrence of the event. An enabling expression is constructed from the values of other signals (possibly, and by default, all signals) that participate in an interface transaction. We believe conditional information is best represented symbolically, as in our scheme, for the following reasons:

- A symbolic representation of interface state eliminates the need to tag each state with an expression that encapsulates signal activity associated with that state.

- A state diagram allows reasoning about the transaction paths of a bus cycle. This is useful, for instance, when resolving multiple reference events.

Khordoc *et al.* and Martello and Levitan do not preprocess an event graph to determine reference events. Consequently, during the verification task they determine the reference event for each pair of events whose temporal relation needs to be verified. We believe this to be computationally expensive, as a typical interface transaction has a few reference events (order of *n*) and many more non-reference events (order of *10n*). It is, therefore, advantageous to determine the reference events and reduce the depth of the event graph to one prior to its use in a verification application.

OEgraph [23, 24] consists of two types of nodes, *operation* and *event*, which are connected by directed arcs to form a bipartite graph. While an event node in an OEgraph represents a class of events, events described in this paper represent discrete events. Timing constraints are specified using first-order predicate calculus (FOPC). We believe that the OEgraph structure may be simplified, and the need for FOPC alleviated, if discrete events are represented on an event graph and state diagrams used to specify sequencing, looping and conditional aspects of interface behavior.

DDS [25] represents timing information, in a timing model, that is a level of abstraction above the signal activity that we represent on an event graph. The timing model essentially captures the manner in which interface timing constraints may influence the scheduling of micro-operations. This model, while perhaps suitable for the integration of timing constraints in the behavioral synthesis process, does not deal with interface behavior at an appropriate level of abstraction.

7. Model Generation

HIDE generates BIMs written in VHDL [26] and Verilog [27][4] code. The modeling style employed by HIDE closely resembles the *process-model graph* structure outlined by Armstrong [3]. A BIM is implemented as a set of processes that execute concurrently. A process is generated for the bus-cycle-level state diagram associated with an interface specification, and for each timing diagram, truth table and bus-state-level state diagram. Code generation for each process is modular and independent of the code generated for other processes. The overall architecture of a HIDE-generated model, using VHDL syntax, is shown in Figure 7.1. As bus-state diagrams and truth tables are optional (they, for instance, are not required for a memory device) their associated processes are not always present in a model.

Each process has a set of signals on its sensitivity list. An event on any of these signals results in the execution of the process. Depending on the status of the input signals on a device during a simulation run, any number of processes may be active at a given time.

Processes communicate with each other through buffers that contain the value driven or received by a signal. A buffer (*<signal-name>_buf*) exists for every signal on the interface of a device. Only processes generated from timing diagrams directly drive or receive values from signals on an interface. This is because temporal aspects of interface behavior are specified only through timing diagrams. When driving a signal, a process sends the value contained in the signal buffer to a signal. When sampling a value on a data or control signal, a process stores the received value in the signal buffer. Processes generated from state diagrams use the buffer value of signals to update the bus state of a cycle. Processes generated from truth tables use and affect the buffer value of signals.

---

[4] In this section, examples of HIDE generated code will be given using VHDL syntax.

```
use libraries.all;
entity <name> is
        generic (
        <link-name> : time := <default-min-value>; /* The minimum value of a timing link
        •                                          at the default operating condition of a
        •                                          device */
        •)
        port (
        <signal-name> : <direction> <type>;
        •
        •
        •)
end name;

architecture interface of <name> is
    type bus_state is (None, s1, s2,...,sn); /* When state diagrams are specified for bus cycles this enumerated type
                                                defines all the bus states of all the cycles in the interface -
                                                None is the default state of a cycle that has no state
                                                diagram*/
    type bus_cycle is (Idle, c1, c2,...,ck); /* For a master device this type defines all the bus cycles */
        signal state : bus_state := None  /* Initial state of a device is None*/
        signal cycle : bus_cycle := Idle; /* Default cycle is Idle;*/
        signal next_cycle : bus_cycle := Idle; /* Default cycle is Idle;*/
        signal pseudo_signal1 : <type>; /* Declaration of all pseudo signals used in the
        •                               interface specification */
        •
        •
        <signal-name>_buf: <direction> <type>; /* Buffers for each signal on a device
        •                                      interface */
        •
        •
        <signal-name>_sample : bit; /* Sample flags for input/bi-directional signals*/
        •
        •
        •
begin
        cycle_transition: process(<sensitivity-list>) /* code for bus-cycle state diagram */
        begin
        •
        •
        •
        end process cycle_transition;
        <cycle-name>_transition: process(<sensitivity-list>) /* code for each cycle that has a bus-state diagram */
        begin
        •
        •
        •

        end process <cycle-name>_transition;
        truth_table_<number>: process(<sensitivity-list>) /* code for each truth table */
        begin
        •
        •
        •
        end process truth_table_<number>;
        <cycle-name>: process(<sensitivity-list>) /* code for each bus cycle */
        begin
        •
        •
        •
        end process <cycle-name>;
end interface;
```

Figure 7.1: Architecture of overall BIM generated by HIDE.

In addition to a buffer the following predefined signals are contained in a model:

- A flag (*<signal-name>_sample*), associated with all input and bi-directional interface ports, that is toggled when the signal is sampled.
- A signal (*state*) that stores the current state of a bus cycle.
- A flag (*trans*) that indicates the occurrence of a state transition.
- A signal (*cycle*) that indicates the current cycle being executed, and another (*next_cycle*) that indicates the next cycle to be executed.

The manner in which interface activity is initiated depends on whether the cycle being executed is a master or slave cycle. A slave cycle's interface behavior is initiated by events on the sensitivity list of its associated process. The interface behavior of a master cycle is explicitly initiated by a user during a simulation run. This may be done through a high-level procedure call that, in turn, drives the corresponding process in a model. Alternatively, a user may cause the execution of a master cycle by forcing values on signal buffers. In either case, the user passes and receives values for data signals through their associated buffers and pseudo signals. Figure 7.2 illustrates an example *command file* used to initiate a read cycle on a MC68020. A command file contains a sequence of instructions used to simulate the interface activity of a device or digital design. The command file is written in pseudo-code; its syntax is dictated by the simulator used.

```
#include mc68020

variable data: integer;

cycle = MC68020;
ADDR_buf = "00";
SIZE_buf = "11";
wait until R_P = 0;
```

Figure 7.2: Example command file.

The following sections discuss the structure of processes generated for timing and state diagrams, and truth tables.

7.1 Processes for Timing Diagrams

Constraint links, as mentioned in Section 5, are not part of the event graph as they take no part in the propagation of timing values. A constraint link is, however, stored with its e_{to} event and points to its e_{from} event. The direction of a constraint link is opposite that of a causal link in an event graph. Code associated with a constraint link checks for a temporal separation between a pair of events. This check can only be made once e_{to} has occurred. Therefore, constraint link code is associated with the e_{to} event.

Figure 7.3 illustrates the event graph shown in Figure 6.3 with constraint links. Code generation examines arcs leading out of each event in the event graph. An event that has no arcs leading out of it has no code generated. An event may have been marked as one that is used by a device to sample a signal or set of signals. This marking is performed using Rule 4.3. In this case, code to implement sampling also needs to be generated for each of the sampled signals. Note that once an event graph has been traversed and annotated with constraint arcs and sampling information, code can be generated for each event by examining its direct links.

Code generation commences with a creation of the sensitivity list for a process. This list consists of signals whose events are in the set of reference events **R** or in the set of events $\mathbf{E_c}$ that have constraint

links pointing to other events. The event e_S, if it is an output event, is excluded from this list. Sequencing events in **R** are replaced by the signal, *trans*, that indicates a state transition. Consequently, as all other reference events are input events (Rule 5.4) and all e_{TO} events for constraint links are also input events (Rule 4.1), the signals on a sensitivity list for a process are all input signals. This is it to be expected as a process should react to signal activity on a subset of the input signals on an interface.

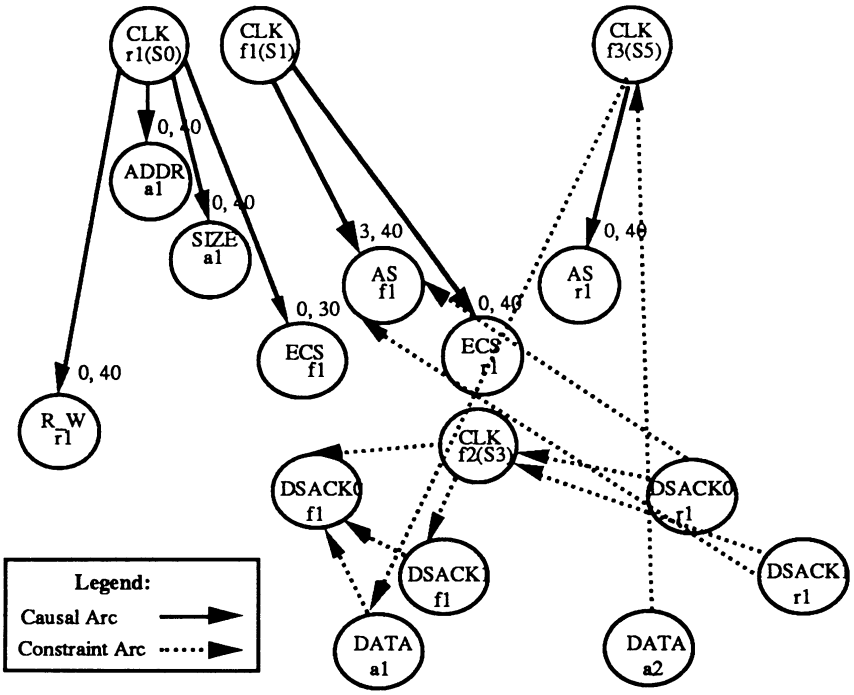

Figure 7.3: Event graph with constraint links.

Each causal arc results in the generation of a *signal-assignment statement* for the non-reference event e_i connected to the reference event e_r in the set **R**. A constraint arc results in the generation of an *assertion statement* that checks the temporal separation between the event e_i that is connected by the arc to an event in the set $\mathbf{E_C}$. An event that is used to sample a signal results in code that stores the value of the sampled signal in that signal's buffer. The value of the corresponding *<signal_name>_sample* signal is toggled to indicate that the signal has been sampled. Each of these statements is nested within a conditional statement that tests for the occurrence of the event from which these arcs fan out. A reference event that is a sequencing event results in a test for the associated bus state, rather than the particular event (Rule 5.5). An illustration of these different statements is shown in Figure 7.4 for the event *CLKf3* associated with bus state *S5* in the event graph shown in Figure 5.1.

Xwave labels an event based on the logic level taken by a signal at the completion of the event. Xwave uses an *event counter* to distinguish between events that have the same label and occur more than once on a signal, for example *CLKf1* and *CLKf3*. A pair of signals (a counter, *<signal-name>_count*, and a flag, *<signal-name>_pulse*), associated with each port on the interface, indicate the number of periodic occurrences of a complementary event pair (for example, rise and fall, active and high-impedance) on the port.

An issue in code generation is when to reset the event counter, *<signal-name>_count*, associated with a signal. This is important because a timing diagram shows the pulse width that needs to be satisfied by a signal through a single pulse on the diagram. During a simulation run, however, the signal may pulse many times during a cycle. Each of these pulses must meet the width constraint. If the event counter for all signals is reset at the end of a bus cycle, then the assertion statement associated with the

pulse-width constraint would not be activated for any pulse other than the first. This is, however, clearly not the intent of the timing specification. To handle this situation, the event counter for a signal is reset when the last event, specified on a timing diagram, for that signal occurs. If a signal with a single pulse was shown on a timing diagram, then the signal's counter is reset when the event that signifies the end of the pulse occurs.

Figure 7.4 shows the code generated for a signal-assignment, assertion and sampling statement, and that related to the manipulation of an event counter. Each signal on a process sensitivity list would have similar code generated to manipulate its event counter. Each event in the sets **R** and $\mathbf{E_C}$ would have code generated similar to that shown for bus state *S5*.

```
READ : process (DSACK0, DSACK1, DATA, trans)  /* trans is a flag used to  indicate change in bus state */
        variable DSACK0_counter : integer := 1; /* event counter for the DSACK0 signal */
        variable DSACK0_pulse : integer := 0; /* counter needs to be incremented at the
        •                                      end of a period (consisting of two transitions) */
        •
begin
    if cycle = 'READ' then
        if DSACK0'event then                    /* code to handle the incrementing of the event
            DSACK0_pulse := DSACK0_pulse + 1; counter associated with the DSACK0 signal */
            if DSACK0_pulse = 3 then
                DSACK0_pulse := 1;
                DSACK0_count := DSACK0_count + 1;
            end if;
        end if;
        •
        •
        •
        if state'event and state = S5 then    /* code associated with the event CLKf3 */
            AS <= transport '1' after T41;    /* signal assignment statement */
            AS_buf <= transport '1';
            assert DATA'last_event > T27   /* assertion statement */
            report "Timing Violation";
            DATA_buf := DATA;          /* code as a result of a sampling arc */
            DATA_sample <= not DATA_sample;  /* used to indicate that the DATA signal has been sampled */
        end if;
        •
        •
        •
        if DSACK0'event and DSACK0 = '1' and DSACK0_count = 1 then
            DSACK0_pulse := 1;          /* reset of the event counter associated with the DSACK0 signal */
        end if;
        •
        •
    end if;
end process READ;
```

Figure 7.4: Portion of a process generated from a timing diagram.

7.2 Processes for State Diagrams

The sensitivity list of a process generated for a bus-cycle-level state diagram contains all the signals that specify the initiation and termination conditions of asynchronous cycles, and also the sequencing signal for synchronous cycles. For example, the sensitivity list for the bus-cycle-level state diagram for a memory device, shown in Figure 2.4, contains the signals *CE* and *WE*.

Code is generated for each transition in the state diagram if the cycle from which the transition is specified *does not* have a bus-state-level state diagram. The code generated is a sequence of conditional statements that capture the information specified by a state-transition table. Each

statement checks the current cycle and the condition associated with the transition. If the transition condition is satisfied then the signal *cycle* has its value updated to the value contained in the *next_cycle* signal. The value *Idle* (in general the name of the default cycle specified by a user) is stored in the *next_cycle* signal. If the cycle to which the transition is made has a bus-state-level state diagram then the start state on this diagram is stored in the signal *state*; if not the value *None* is stored in *state*. If the cycle to which a transition is made is a master cycle then the signal *R_P* (*Request Pending*) is set to 1. The specification for the reset of this signal is assumed to have been specified through a truth table or on a timing diagram (see Figure 2.7).

A procedure, in a command file, that causes the execution of a master cycle waits for *R_P* to be reset before terminating and allowing the next procedure to be executed. *R_P* may be reset during the execution of a cycle, allowing the next procedure in the command file to initialize buffers and commence execution without intermediate *Idle* cycles. The signal R_P, therefore models a signal that informs internal circuitry about the status of execution of a cycle and affects the scheduling of these cycles.

Figure 7.5 shows a portion of the code generated for the bus-cycle-level state diagram for the MC68020 shown in Figure 2.5. Similar code would be generated for each transition in the state diagram as outlined earlier in this section.

```
CYCLE_TRANSITION : process (CLK)
begin
    if CLK'event and CLK = '0' and next_cycle = Read and cycle = Idle then
        cycle <= Read
        state <= S0;
        next_cycle <= Idle;
        R_P <= '1';
    end if;
    •
    •
    •
end CYCLE_TRANSITION;
```

Figure 7.5: Portion of a process generated from a bus-cycle level state diagram.

The sensitivity list for a process generated from a bus-state-level state diagram contains only the sequencing signal. For the state diagram in Figure 2.6 the sequencing signal is the *CLK* signal. Each statement checks the current bus state and conditional information specified for transitions associated with the bus state. If a transition condition is satisfied then the state is updated to its new value. Also, the *trans* signal is toggled to indicate that a state transition has been made. For a transition that indicates the completion of a cycle as many conditional statements are generated as arcs leading out from the cycle at the bus-cycle-level state diagram. Each of these conditional statements evaluate the initiation condition for cycles reachable from the current cycle, in addition to the condition associated with the transition at the bus-state level. The actions associated with a transition that terminates a cycle are identical to those generated for a transition at the bus-cycle level. The process generated for a portion of the state diagram in Figure 2.6 is shown in Figure 7.6.

7.3 Processes for Truth Tables

The sensitivity list for a process generated from a truth table consists of sampling flags associated with the following signals:

1. Those signals on the LHS of a truth table that are input to a device.

2. Those signals on the RHS of an action associated with a row in a truth table.

These signals represent *data flow* constraints on the actions in a truth table. Recall that a flag for a signal is set when that signal is sampled by a process generated from a timing diagram. When all sampling flags associated with a process generated from a truth table have been set, then the actions associated with a row in the truth table are performed. The code generated for each row of the truth table is one that tests whether the row conditions are satisfied, and if so executes the actions associated with that row. A signal whose value is X for a row in the truth table does not form part of the conditional statement. The process generated for a portion of the truth table in Figure 2.7 is shown in Figure 7.7.

```
READ_STATE_TRANSITION : process (CLK)
begin
    if cycle = Read or cycle = Write then
        if CLK'event and CLK = '0' and state = S0 then
            state <= S1;
            trans <= not trans;   /* indicate that a state transition has been made */
        end if;
        •
        •
        •
    end if;
end READ_STATE_TRANSITION;
```

Figure 7.6: Portion of a process generated from a bus-state level state diagram.

```
TRUTH_TABLE_1 : process (DSACK0_sample, DSACK1_sample, DATA_sample)
        variable DSACK0_sample_flag: integer := 0;   /* flags used to indicate whether all signals
        variable DSACK1_sample_flag: integer := 0;      that are a part of the data flow constraint
        variable DATA_sample_flag: integer := 0;        of a truth table have been sampled */
begin
        if DSACK0_sample'event then
            DSACK0_sample_flag := 1;
        end if;
        •
        •
        •
        if DSACK0_sample_flag = 1 and DSACK1_sample_flag = 1 and DATA_sample_flag = 1 then
            if R_W = '1' and SIZE(1) = '0' and SIZE(0) = '0' and ADDR(1) = '0' and ADDR(0) = '1' and DSACK0 = '0' and
            DSACK1 = '0' then
                DATUM(31 downto 0) := DATA_buf (23 downto 0);
                ADDR_buf := ADDR_buf + 3;
                SIZE_BUF := SIZE_BUF - 3;
            end if;
            •
            •
            •
        end if;
end TRUTH_TABLE_1;
```

Figure 7.7: Portion of a process generated from a truth table.

8. Related Work

In this section, we contrast our approach to model generation and interface specification with that of others, and also point out the limitations of SpecIT and HIDE.

General approaches have been taken toward the automated generation of simulation models. HUM [28] is a commercial tool that generates VHDL models representing the behavior of both the computation and interface engine of a device. HUM requires a spreadsheet-like specification of the

behavior of a device. XBIF [29] is a user interface that provides both graphical and tabular specification formats of device behavior, which may then be converted to VHDL or retained in an internal [30] representation for synthesis tasks. Both XBIF and BIF rely on the specification and representation, respectively, of behavior using hierarchical annotated state tables. While the use of state tables is appropriate for specifying control flow, we believe that timing diagrams are necessary to capture the temporal behavior that is an integral part of an interface specification.

Research in the area of interface specification has focused on interface timing verification and synthesis tasks, such as the synthesis of interface adapters and the assimilation of interface constraints during the synthesis of the internal circuitry of a device.

Textual approaches to timing or interface specification have (a) used temporal logic [31], (b) extended the features of an HDL to allow for the specification of timing constraints [32, 33, 34], and (c) used the features of a software programming language (SPL) to express timing constraints [35].

Nestor and Thomas [32, 33], and Camposano and Kunzmann [34] take a similar approach to extending ISPS and DSL (two HDLs), respectively, to accomodate timing constraints in behavioral-synthesis. While Camposano and Kunzmann do not capture interface timing constraints, Nestor and Thomas do. The interface specification in BSI (extended ISPS) is not distinct from the internal-circuitry specification; rather, a composite specification is used to guide the synthesis process. We believe, instead, it is desirable to have a modular specification of internal and interface behavior. Also, both BSI and DSL only permit the specification of timing *constraints*; the causal information that is contained in our specification is a *result* of synthesis tasks, and represents the delay associated with the generation of output signals.

The disadvantages of textual specifications of interface behavior are the following:

- They require knowledge of a programming language (hardware or software), or specification style such as temporal logic that is not a natural means for the specification of hardware behavior to designers. Designers conceptualize hardware in terms of timing, state diagrams and truth tables [4].

- They often comingle the specification of internal and interface behavior.

Graphical specification tools have been developed by Borriello [7, 12] and Subrahmanyam [13, 14].

Waves [7, 12] is a tool used for the capture of interface specifications for the synthesis of interface adapters. Waves uses a notion of *formalized timing diagrams*, which extend conventional diagrams by: (a) allowing signal values to be specified as a function of other signals, (b) using regular expressions to specify the transaction paths of a bus cycle, and (c) allowing timing diagrams to be interconnected by linking events in different diagrams. SpecIT has more embedded knowledge about the structure of a timing diagram and semantics that may be culled from structural elements. This allows a user to enter a diagram in a uniform manner without having to explicitly provide semantic information associated with elements of a timing diagram. Also, we have defined rules that formalize the representation of interface behavior using event graphs. These specification and representation rules allow formal reasoning on interface behavior.

Subrahmanyam [13, 14] developed a tool for the capture of timing diagrams whose information is represented internally by a format built on temporal-logic constructs, and used subsequently for the synthesis of mixed-mode (asynchronous and synchronous elements) systems. Subrahmanyam, however, does not attempt to capture all the information in a graphical form; information not easily captured on a timing diagram is specified textually.

8.1. Limitations of SpecIT and HIDE

Our interface specification methodology currently allows three hierarchical specification levels (bus-cycle-level, bus-state-level, and signal activity). It may be desirable to allow the specification of hierarchical state diagrams with arbitrary levels of nesting [29] for the following two reasons:

- To ease the specification of a complex interface with many bus states.
- To permit the specification of causal and constraint information at multiple hierarchical levels.

It may also be useful to allow timing diagrams to be linked to specify signal activity associated with multiple transaction paths.

We make an assumption that all asynchronous bus cycles have a single transaction path and consequently cannot specify or model asynchronous cycles with multiple transaction paths.

9. Results

HIDE has been used to generate models for the Intel I80386, Motorola MC68020 and MC68332 CPUs, a Cypress Semiconductor memory device, CY7C147, and other non-commercial interface specifications that exercised the capabilities of the tool. The model generation time using HIDE is a few man-days, in contrast to a few man-months when the same task is performed manually. Most of the time is spent in understanding device interface behavior; the execution time of HIDE is of the order of a few seconds. Figure 9.1 summarizes some of the relevant results of the model generation process[5] .

Note that the MC68332 required fewer man days to generate a model because of its similarity to the MC68020. Much of the effort spent in understanding and specifying the MC68020 was applicable when generating a model for the MC68332.

The models accurately simulated device behavior. Particular attention was paid to the ability of the models to handle dynamic bus-sizing, multi-cycle interface transactions and the execution of consecutive master cycles without intermediate *Idle* cycles (when so desired).

DEVICE	GRAPHICAL INPUTS			LINES OF CODE	MAN DAYS
	STATE DIAGRAMS	TIMING DIAGRAMS	TRUTH TABLES	(VHDL)	
MC68020	3	6	2	1134	7
MC68332	4	8	2	1368	3
I80386	2	3	4	1051	10
CY7C147	1	2	0	332	1

Figure 9.1: Results of the Application of SpecIT and HIDE

[5] Verilog models were, on average, 20% more verbose than VHDL models.

10. Summary

In this paper we have presented a system, HIDE, for the automated generation of BIMs from a high-level specification of interface behavior using constructs that are familiar to a hardware engineer. We have provided a methodology for the specification of interface behavior, and built a tool, SpecIT for this purpose. SpecIT captures asynchronous and synchronous behavior in a uniform manner. Our methodology for interface specification conforms to the criteria outlined earlier in the paper and is consistent with standard logic design methodologies.

We have identified the semantics associated with structural elements of an interface specification, and have shown how knowledge of this structure facilitates the automated generation of simulation models. The notion of causal and constraint links, and the ability to identify these links in a timing diagram significantly enhances our representation of timing information using event graphs. The ability to identify the occurrence of a sampling event, specified implicitly on a timing diagram, allows a simple methodology for interface specification whose components are modular and complementary. The notion of a range of occurrence for an event, the rules it adheres to, and the manner in which it is computed have been formally defined. These are important formalisms associated with an event graph and facilitate the representation of temporal information.

Acknowledgements

Yew-Hong Leong developed the first Xwave/HIDE system and many of the ideas in this paper stem from that work. We thank Anurag Gupta for his insightful comments.

References

[1] Logic Automation, Inc. Processor Control Language, Users Guide.

[2] D. Coelho. The VHDL Handbook. *Kluwer Academic Publishers*, 1989.

[3] J. Armstrong. Chip-Level Modeling with VHDL. *Prentice-Hall Inc.*, 1989.

[4] W.I. Fletcher. An Engineering Approach to Digital Design. *Prentice-Hall, Inc.*, 1980.

[5] M. McFarland. CPA: Giving an Account of Timed System Behavior. In *Proceedings of TAU'90*, August 1990.

[6] W.P. Birmingham and Yew-Hong Leong. The Automatic Generation of Bus-Interface Models. In *Proceedings of the 29th Design Automation Conference*, June 1992.

[7] G. Borriello. A New Interface Specification Methodology and its Application to Transducer Synthesis. *Technical Report UCB/CSD 88/430 (PhD Dissertation)*, Computer Science Division, University of California at Berkeley, May 1988.

[8] H.S. Stone. Microcomputer Interfacing. Addison-Wesley Publishing Company, February 1983.

[9] Motorola, Inc. MC68020 32-bit Microprocessor User's Manual. *Prentice Hall Inc.*, 1990.

[10] P. Rony. Interfacing Fundamentals: Timing Diagram Conventions. *Computer Design*, January 1980.

[11] A.R. Martello, S.P. Levitan and D.M. Chiarulli. Timing Verification using HDTV. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[12] G. Borriello. Specification and Synthesis of Interface Control Logic. In High-Level VLSI Synthesis, edited by Raul Camposano and Wayne Wolf. *Kluwer Academic Publishers*, 1991.

[13] P. A. Subrahmanyam. Automated Synthesis of Systems with Interacting Asynchronous (Self-Timed) and Synchronous Components. In *Proceedings of the International Conference on Computer Design*, October 1989.

[14] P. A. Subrahmanyam. TALES: Event-Based Semantics for Timing Specification. (with Application to Synthesis, Verification and Analysis). In *Proceedings of TAU'90*, August 1990.

[15] F. Mavadatt and T. Gahlinger. On Deducing Tight Bounds from Partial Timing Specifications. In *Proceedings of TAU'90*, August 1990.

[16] A.R. Martello and S.P. Levitan. Temporal Specification Verification Via Causal Reasoning. In *Proceedings of TAU'92*, February 1992.

[17] D.Ku and G. De Micheli. Relative Scheduling under Timing Constraints. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[18] D.E. Wallace and C. H. Sequin. ATV: An Abstract Timing Verifier. In *Proceedings of the 25th Design Automation Conference*, June 1988.
[19] J.F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM 26(11)*, 1983.
[20] K. Khordoc, E. Cerny, and M. Dufresne. Modeling and Execution of Timing Diagrams with Optional and Multi-Match Events. In *Proceedings of TAU'92*, February, 1992.
[21] K. Khordoc, M. Dufresne, and E. Cerny. A Stimulus/Response System Based on Hierarchical Timing Diagrams. In *Proceedings of the International Conference on Computer Aided Design*, December, 1991.
[22] A.R. Martello and S.P. Levitan. Causal Timing Verification. In *Proceedings of TAU'90*, August, 1990.
[23] T. Amon and G. Borriello. On the Specification of Timing behavior. In *Proceedings of TAU'90*, August, 1990.
[24] T.Amon, G. Borriello and C. Sequin. Operation/Event Graphs: A Design Representation for Timing Behavior. In *Proceedings of the Tenth International Conference on Computer Hardware Description Languages and their Applications (CHDL '91)*, IFIP April 1991.
[25] S. Hayati, A. Parker and J. Granacki. Representation of control and timing behavior with applications to interface synthesis. In *Proceedings of the International Conference on Computer Design*, October 1988.
[26] Institute of Electrical and Electronics Engineering, *IEEE Standard VHDL Language Reference Manual, IEEE Std. 1076-1987*, 31 March 1988.
[27] D.E. Thomas and Philip Moorby. The Verilog Hardware Description Language. *Kluwer Academic Publishers*, 1991.
[28] Lewis System, Inc. HUM Users' Manual, VHDL PC Version, July 1991.
[29] N. Dutt, J. Cho and T. Hadley. A User Interface for VHDL Behavioral Modeling. In *Proceedings of the Tenth International Conference on Computer Hardware Description Languages and their Applications (CHDL '91)*, IFIP April 1991.
[30] N.D. Dutt, D.D. Gajski and T. Hadley. BIF: A Behavioral Intermediate Format for High Level Synthesis. *ICS Technical Report 89-03*, University of California at Irvine, February, 1989.
[31] G.V. Bochmann. Hardware Specification with Temporal Logic: An Example. *IEEE Transactions on Computers*, March, 1982.
[32] J.A. Nestor and D.E. Thomas. Behavioral Synthesis with Interfaces. In *Proceedings of the International Conference on Computer Aided Design*, December 1986.
[33] J.A. Nestor. *Specification and Synthesis of Digital Systems with Interfaces*, PhD Thesis, Carnegie Mellon University, April, 1987.
[34] R. Camposano and A. Kunzmann. Considering Timing Constraints in Synthesis from a Behavioral Description. In *Proceedings of the International Conference on Computer Design*, October 1986.
[35] E.F. Girczyc, R.J. Buhr, and J.P. Knight. Applicability of a Subset of Ada for Graph-Based Hardware Compilation. *IEEE Transactions on Computer Aided Design*, April, 1985.