Chapter 9

Synthesis Related Aspects of Simulation

Franz J. Rammig

9.1 Introduction

The starting point of a chapter on synthesis related aspects of simulation has to be the assertion that, at least in theory, as any other verification method simulation becomes obsolete by synthesis, because one intent of using synthesis is to produce designs that are correct by construction. So what can simulation be good for in such a context?

First of all, one has to make the distinction between *fully automatic* and *interactive* synthesis. No doubt, some kind of verification is needed for interactive synthesis, to check the manipulations the user imposes on the design. In Chapter 8 formal methods for this purpose are discussed. In contrast to those "analytical methods," in this chapter "experimental methods" are explained. But, one has to keep in mind that simulation is not *in*formal at all. Both approaches, formal verification and simulation, have benefits as well as restrictions and both methods are heavily dependent on intelligent usage. Thus, in this chapter one main emphasis is on how to use simulation properly.

Under the assumption of fully automatic synthesis, one may doubt whether there is any need for verification, especially for simulation, at all. However, using synthesis does not affect the correctness of an initial specification. Since specification is the first formal document, it can only be proven to be consistent and in accordance with certain fixed or generic rules. By no means can it be verified analytically whether or not the initial specification matches the designer's intent. Because synthesis even in the ideal case can guarantee nothing more than an implementation which is correct in relation to a given specification, the term *correctness by construction* has a meaning that reduces correctness to correctness modulo specification. It can be concluded, therefore, that independent of the synthesis strategy (fully automatic or interactive) and independent of the availability of analytical (formal) methods, the initial specification has to be verified by simulation. Using simulation, intelligent experiments have to be carried out to ensure that the specification is in accordance with the designer's intention.

In this chapter, the term "simulation" is used slightly differently from the traditional usage. Simulation, in general, is used to validate that design steps have been carried out correctly. Consequently, simulation systems in the past have been optimized for lower levels of abstraction (logic level or circuit level). When used in the context of synthesis, simulators are needed at higher levels, both for checking adequateness and performance. Most important, however, in the case of synthesis the designer has to be supported validating his or her initial specification. This implies that planning, performing, and analysis of intelligent experiments have to be supported. This is an aspect missing in most traditional simulation systems.

The question remains about the relevance of simulation at lower levels of abstraction. As already mentioned, it is needed for interactive synthesis, competing—or better cooperating—with analytical methods. In the case of fully automatic synthesis it is, in principle, obsolete. However, designers tend to distrust synthesis systems. Of course, like any other software, synthesis systems are not free of "bugs." Therefore, low-level simulation offers an opportunity to crosscheck the synthesis results. Another argument for low-level simulation can be seen in *synthesis for simulation*, when a hardware specification, unknown to be correct, may quickly be translated into a low-level implementation and efficiently be simulated at this level of abstraction. The behavior at the lower level, then, has to be retranslated to the higher level of abstraction. In the extreme case, the low-level simulation may be performed on a so-called *simulation engine* (hardware accelerator), which has a multiprocessor architecture and whose instruction set is optimized for the execution of logic-level simulation, i.e., synthesis can be used as a compiler frontend for the respective simulation engine. This method has its analogy in software design, where it has been observed that low-level instruction sets allow a more efficient execution of high-level programs (after compilation) than architectures that support the high-level languages directly. Therefore, experiments (tests) with high-level programs are carried out after compilation at the lower level, and only the diagnostics are retranslated to the higher level of abstraction.

The third argument for simulation of synthesized designs at lower levels of abstraction derives from the fact that the transformation from high- to low-level descriptions provided by synthesis is by no means unique. Synthesis results can vary greatly in low-level characteristics, such as delays and power consumption, which are not or only incompletely specified in the input description. Low-level simulation can be used in this situation to evaluate the quality of synthesized implementations.

To conclude, simulation is needed within the context of synthesis, both for automatic and interactive synthesis. There are good reasons to consider simulation at high and low levels of abstraction. Simulation systems, therefore, have to support the planning, execution, and analysis of intelligent experiments.

9.2 Multi-Level Modeling

9.2.1 Levels of Abstraction

As described in Chapter 1, a widely accepted scheme of levels of abstraction and domains of description is used in the area of digital system design. In the context of simulation, the behavioral domain—especially the modeling of time and data—is of main interest.[1]

At the *circuit level* behavior of electronic circuits built from resistors, capacitors, etc., is modeled over the time axis. This is achieved by a system of differential equations, i.e., both the time axis and the observable data (currents, voltages) are represented by a continuous value domain. There are some simulators at this level of abstraction, with SPICE [Nage87] in numerous versions being the most widely used.

The *logic level* has a long tradition in digital system design. This level has solid mathematical foundation in Boolean algebra. However, with Boolean algebra only the timeless behavior can be modeled. Some additional concepts have to be considered in order to cover time as well. In the ideal case the value domain[2] is restricted to Boolean values 0 and 1 while the time domain remains continuous, but the problem of uncertain values necessitates the introduction of additional values. As a consequence, in most cases the underlying algebra is no longer Boolean. The operators, however, are always Boolean (logic) operators. Because of the long tradition of using the logic level for simulation purposes, numerous simulators are available for this level of abstraction. They differ in

[1]A level of abstraction, which plays a major role in simulation, located between circuit and logic level, is the switch level, where circuits are modeled as networks of ideal switches and capacitors. As this level plays no role in synthesis, it is not further discussed here, and is not included into the representation of the Y-chart of Chapter 1.

[2]The term "domain" is used here in another sense than in the context of the Y-chart where it denotes the view or aspects of a design object one is interested in. Here, domain refers to a range of values.

their definition of value domains and timing models. HILO [HILO91], CADAT [CADA91], DISIM [DISI91] are some examples of commercially availabl simulators.

At the *register-transfer level* a specific mode of operation is assumed. Components continuously observe specific, associated conditions. Whenever the condition associated with a component becomes true, this component performs its specific operation. Operations can be interpreted as a transfer of data between registers; the data transferred may be modified during this operation. The value domain at this level of abstraction is given by (uninterpreted) bitstrings, whereas the timing model is that of counting clock ticks. Thus, the time domain has become discrete as well. The register-transfer level is very helpful for well-structured synchronous designs, since it enforces to some extend such a design style. Simulation at this level of abstraction has been mainly studied in academic institutions [Borr81, Chu74, DuDi75, Hart77].

At the *algorithmic level* the reactive point of view of the register-transfer level is inverted to an imperative one. While at the register-transfer level the system is seen from the point of view of the individual components, at the algorithmic level the controller's point of view is taken. In contrast to ordinary algorithmic descriptions, however, concurrency plays an important role in hardware design and, therefore, at this level of abstraction. In contrast to the register-transfer level, where it is precisely specified which conditions trigger an operation, at the algorithmic level this information is not explicitly provided. Only the discrete point of time at which an operation has to be carried out, is identified. The domain of values can be defined in various ways, but is usually restricted to bitstrings with interpretations attached (e.g., two's complement integer, IEEE standard floating point number, ASCII character). The timing model at this level of abstraction is either that of counting clock ticks as at the register-transfer level or a purely causal one. In the latter case, simply a causality structure is assumed as it is known from general programming languages. Only a few commercial simulators are available for this level of abstraction [DACA90, Thom91].

Finally, at the *system level* the entire system is considered as a set of cooperating processors. In this context the term processor is used in a broader sense, namely to denote a subsystem which exports certain services (instructions). Both the value domain and the timing model are purely symbolic at this level of abstraction. Types can be defined freely and associated with arbitrary semantics. The sequence of operations is only defined by causality relations. The system level is supported by only very few commercial simulators. More simulators working at this level of abstraction will evolve as synthesis approaches this level of abstraction.

9.2.2 Modeling Concepts

In the context of synthesis three major modeling concepts have to be considered by a simulation system: imperative models, reactive models, and stimulated equations (strictly functional models).

Imperative models originate from algorithmic programming languages for von Neumann type processors. High-level synthesis in most cases starts from a high-level description in imperative style (cf. Chapter 6). Therefore, the support of this model is essential in the context of this book. While in traditional imperative programming languages a strictly sequential operation is assumed, this concept has to be generalized to concurrent algorithms in the domain of hardware design. This is necessary, because hardware systems are concurrent by nature, i.e., performance is gained by the use of parallelisms. An obvious way for generalizing sequential algorithms to concurrent ones is the approach of communicating sequential processes (CSP) [Hoar78, Hoar85]. Therefore, this model may serve to introduce this concept.

In CSP the entire system is represented as a system of concurrently active processes, where each process is strictly sequential. This concept, perfectly implemented in the programming language OCCAM [May83], can also be identified in VHDL (cf. Chapter 2). In CSP, a single process is represented using the constructs *assignment*, *guarded command*, and *iteration*. The processes are completely independent, i.e., they are not allowed to share any resources other than communication channels. In this aspect, VHDL is less strict, because shared resources are allowed. Any conflicts, however, have to be resolved statically using *resolution functions* in VHDL.

Events and processes. Objects to be specified are described using *events*. Such an event is treated as an atomic action. Typically, an event is the assignment of a value to a variable. The set of all events used to describe an object is called *alphabet* of this description (of this object). A *process* is an arbitrary behavior of an object that can be expressed using the alphabet of this object.

Sequential execution. Let a be an event and P a process. By "**seqbegin** a; P **seqend**" it is denoted that event a has to happen first, before process P can be started. By definition, a is a process, and if P is a process, "**seqbegin** a; P **seqend**" is a process, too.

Recursion. Let P be a process. By "**while** true **do** P" it is denoted that P has to be repeated infinitely. If P is a process, "**while** true **do** P" is a process, too. Let *con* be a binary variable. By "**while** con = true **do** P" it is denoted that P has to be executed as long as *con* has the value "true." If P is a process, "**while** con = true **do** P" is a process, too.

Case distinction. Let $P_1, \ldots, P_n$ be processes, *cnt* a variable with domain $\{c_1, \ldots, c_n\}$. By "**case** *cnt* **of** $c_1 : P_1; c_2 : P_2; \ldots; c_n : P_n$ **caseend**" it is denoted that only that P_i has to be executed whose c_i is the current value of *cnt*. If $P_1, \ldots, P_n$ are processes, "**case** *cnt* **of** $c_1 : P_1; c_2 : P_2; \ldots; c_n : P_n$ **caseend**" is a process, too.

Input/output. Let *chan* be a special variable of type **channel**, denoting a communication channel. Let *var* be an arbitrary variable. By *chan!var* an output operation is denoted. The actual value of *var* is transmitted via channel *chan*. The operation is not completed until a concurrently active process (see below) has read this value (rendezvous concept). By *chan?var* an input operation is denoted. The actual value taken from the channel *chan* is assigned to *var*. This operation can be carried out only if the channel is not empty. Initially, a channel is empty and a non-empty channel is emptied by executing an input operation on it. Like assignments, input/output operations are treated as events. By definition, neither an input operation nor an output operation can be executed if no concurrently active additional process exists. It is assumed that a function **test***(chan)* exists for each channel *chan*. It returns **true** if *chan* is not empty, **false** otherwise. This function can be used only within control expressions and is not treated as an input operation (does not empty a channel and does not block on an empty channel).

Sequential process. A *sequential process* is an arbitrary sequence of the above constructs, and nothing else.

Concurrent process. Each sequential process is also a *concurrent process*. Let P be a sequential process and C a concurrent one. By "**conbegin** P; C **conend**" it is denoted that P and C are executed concurrently, i.e., P and C are initiated at the same point of time and, then, run completely independent as long as they do not communicate via a shared channel. The process "**conbegin** P; C **conend**" terminates when the last one of the contained processes P and C terminates. As mentioned above, processes within a concurrent process are not allowed to share resources other than channels. If P is a sequential process and C is a concurrent one, "**conbegin** P; C **conend**" is a concurrent process, too.

Locally, VHDL has a similar point of view. However, internally all processes are strictly sequential and are treated as looping infinitely, the loop being interrupted in each cycle until events specified in the *sensitivity list* of the process occur. Therefore, VHDL follows a reactive modeling concept.

With the imperative model, a system is observed from the controller's point of view. In this sense, a controller is an object that causes other objects to execute operations in a well defined (partial) order. The reactive model inverts this point of view. In this case, the entire system is observed from the position of the controlled objects. From this (local and partial) point of view the partial ordering of the operations has no meaning. For any specific object, it is relevant only that a certain action has to be executed whenever a certain condition becomes true. Such an action may include a modification of certain conditions, thus implicitly causing other objects to execute operations. The descriptive power of the reactive model is the same as that of imperative modeling. However, the global control concepts of a description are no longer visible. The following simple example may help to illustrate this (VHDL notation):

```
process begin
        operation_0;
        operation_1;
        operation_2;
end process
```

This is equivalent to

```
signal sequence: Natural := 0;
-- auxiliary signal, initialized to 0
process begin
        operation_0;
        sequence <= 1;
        wait until sequence = 0;
end process;
process begin
        operation_1;
        sequence <= 2;
        wait until sequence = 1;
end process;
process begin
        operation_2;
        sequence <= 0;
        wait until sequence = 2;
end process;
```

After the start of the system all three processes are continuously active. Whenever the condition "sequence = i" becomes true the respective process is (re-) started and the two actions "operation_i" and sequence "$\Leftarrow ((i+1) \bmod 3)$" are executed.[3] The sequence in which the processes are written down in the descriptive text is of no influence.

The reactive model seems to be quite natural for hardware descriptions, because it takes a structural point of view with behavior attached, and it is the basic modeling concept at the register-transfer level. At this level of abstraction, the basic operation is the storage of information (possibly after modification) into a destination register under certain conditions. This is nothing else than inverting the point of view of microprogramming, or looking at a microprogram with the controlled object's eyes. The global semantic concept of VHDL is, as such, a reactive one.

What happens if the conditions in a reactive model are always true? In this

[3]Due to technical reasons, VHDL requests a resolution function to be defined for the signal sequence. For reasons of readability, this function is omitted here.

case, the attached operations have to be executed continuously. Restricting reactive models to those for which every condition is always true, does not restrict the modeling power. Since, obviously, logic-level implementations exist of all digital systems described at the RT-level, this may not be too surprising. The same can be shown within the modeling context as well:

A reactive model is represented by a set

$$R = \{\textbf{at } c_i \textbf{ do } t_i := f_i(s_i) | i = 1 : n\}$$

, i.e., a set of "guarded commands" with the meaning that whenever c_i becomes true, the value calculated by function f_i on its arguments s_i is assigned to t_i. It is not requested per se that the destination ranges t_i are disjoint, but it is always possible to rewrite R in such a way that this becomes true. In this case, for each destination object the sequence of its values is defined. On the other hand, each f_i can be modified in such a way that the executability condition c_i becomes an argument of this function:

$$f_i'(s_i, c_i) := \begin{cases} f_i(s_i) & \text{if } c_i = \text{true} \\ t_i & \text{else} \end{cases}$$

But then, the prefix "**at** c_i **do**" becomes obsolete and can be omitted. By the procedure described above a system of equations is obtained. In the stable state all equations contained are in equilibrium (the system of equations is solved). By changing the value of an arbitrary object, the equilibrium is distorted, resulting in an instable global state. As a reaction, the system of equations trys to restabilize (recalculate a solution). Of course, the existence of a global stable state (a solution) is not always guaranteed. In that case, the system will continuously attempt to reach a stable state, without success.

In the above it is assumed that the left-hand side and the right-hand side have different meanings (destination and source). In this case, a system of unidirectional value assignment is modeled. In a system like this a well-defined flux of distortions through the system exists, similar to a wavefront. If there are no different roles, the assignment symbol becomes an equality sign. Of course, the flux of distortions then becomes much more complex. Both models, unidirectional and bidirectional flux of distortions make sense in hardware modeling. This kind of functional modeling is the most natural one at lower levels of abstraction, as well. Unfortunately, it is not at all supported by VHDL.

9.2.3 Modeling of Time

To model the timing of a system is of crucial importance, even though it adds to the complexity of simulation. In addition, the problem exists that in real systems a value at a certain point of time is not only dependent on calculations based upon arguments at exactly the same point of time, but on a history of values over the time axis. In the range of abstractions to be considered in this context time dependencies can be reduced to delays, which are either inertial or inertial free delays. This concept can be introduced quite easily at the logic level and can be generalized to other levels of abstraction, afterwards. The following discussion is based on the assumption that the observable values are modeled by a set of five elements $\{1, 0, p, n, u\}$ where 1 and 0 stand for stable values, n and p for negative or positive edges, and u for an uncertain value [Ramm80a]. This value model is used in this section for simplicity reasons, only; for more advanced models the reader is referred to [Coel89, Dani70, Haye86, KrAn90, LeRa83].

Definition 9.1 (Real continuous signal set, RCS) *Let $[L, R] \subset \mathbf{R}$ be a real interval. A set of differentiable real functions $RCS \subset [L, H]^{\mathbf{R}}$ is called real continuous signal set, $rcs \in RCS$ real continuous signal. L and H stand for the lowest and highest voltage available, for example, in the particular technology.*

This infinite set of observable values has to be mapped onto a set of only five elements.

Definition 9.2 (Five valued continuous signal set, $FV^{\mathbf{R}}$) *Let $H > TH > TL > L$ be reals, $FV := \{0, p, n, 1, u\}$. $FV^{\mathbf{R}}$ is called five valued continuous signal set, $fcs \in FV^{\mathbf{R}}$ is called model of a $rcs \in RCS :\Leftrightarrow fcs := f(rcs)$ with*

$$\begin{aligned}
\forall t \in \mathbf{R} : \; & f(rcs_t) = 1 :\Leftrightarrow rcs_t \geq TH \\
& f(rcs_t) = 0 :\Leftrightarrow rcs_t \leq TL \\
& f(rcs_t) = p :\Leftrightarrow TH > rcs_t > TL \;\wedge\; drcs_t/dt > 0 \\
& f(rcs_t) = n :\Leftrightarrow TH > rcs_t > TL \;\wedge\; drcs_t/dt < 0 \\
& f(rcs_t) = u :\Leftrightarrow \text{otherwise}
\end{aligned}$$

To achieve a further abstraction, the time set is restricted to a discrete one:

Definition 9.3 (Five valued discrete signal set FV^T) *Let T be a countable subset of $\mathbf{R}$ with metric and ordering inherited from $\mathbf{R}$. FV^T is called five valued discrete signal set, $fds \in FV^T$ five valued discrete signal.*

Let $\ldots, t_{i-1}, t_i, t_{i+1}, \ldots$ denote adjacent elements (points of time) of T. $fds \in FV^T$ is called model of a $fcs \in FV^{\mathbf{R}} :\Leftrightarrow fds = f(fcs)$ with:

$$\begin{aligned}
& \forall t_i \in T : \forall w \in \{0, p, n, 1\} : \\
& \qquad f(fcs_{t_i}) = w :\Leftrightarrow \forall t_{i-1} < t \leq t_i : fcs_t = w \\
& \qquad\qquad\quad = u \text{ otherwise}
\end{aligned}$$

With the above abstraction, signals can be represented by traces [Snep85]. For instance, a 50 MHz clock signal with a transition time of 4 ns for the raising edge and 6 ns for the falling edge is modeled by the sequence[4]

$$(pppp1111nnnnnn00000)^*.$$

It is a basic principle for the modeling of real time behavior to distinguish between purely functional aspects and timing.

Definition 9.4 (Timeless Boolean function) *Let a_t denote the value of a signal at point of time t ("projection at t"), as already used above. $h : (FV^T)^n \rightarrow FV^T$ is called timeless Boolean function*
$:\Leftrightarrow$
$\exists(h' : FV^n \rightarrow FV) : \forall a \in (FV^T)^n : \forall t \in T : (h(a))_t = h'(a_t)$

The purely functional aspects are modeled perfectly by Boolean functions. In addition, timing effects have to be considered:

a) **Delay:** It is assumed that a value calculated by an operation comes to effect only some time period after the causing arguments have been applied. No distortion of the waveform takes place.

b) **Inertia:** Usually, switching elements react only to signals that are stable for a specific minimal period of time.

c) **Edge triggering:** Certain switching elements react on transitions rather than on stable values. Usually, a minimal slope requested for the transition has to be accepted.

d) **Transition time modification:** Switching elements may either decrease the slope of transformed signals, or act as pulse sharpeners.

In the following, a number of elementary time dependent functions are introduced. They can be used to compose realistic models of switching elements.

Definition 9.5 (Transport delay function)
Let $TD := (t_{PLH}, t_{PHL}) \in \mathbb{N}_0^2, t_{PLH} + t_{PHL}$ finite, a transport delay specification, $md := max\{t_{PLH}, t_{PHL}\}$.

The function $tdf_{TD} : FV^T \rightarrow FV^T$ is called transport delay function with respect to TD
$:\Leftrightarrow$
$\exists(tdf' : FV^{md} \rightarrow FV) : \forall a \in FV^T : \forall t \in T :$
$\quad (tdf_{TP}(a))_t = tdf'(a_t, \ldots, a_{t-md})$ *with*

[4] "*" denotes repetition.

$$tdf'(a_t,\ldots,a_{t-md}) = \begin{cases} a_{t-t_{PLH}} & \text{if } a_t \in \{1,p\} \vee (a_t = u \wedge t_{PLH} \geq t_{PHL}) \\ a_{t-t_{PHL}} & \text{otherwise} \end{cases}$$

Transport delay is offered by VHDL, but the distinction between t_{PLH} and t_{PHL} (between positive and negative transitions) is not supported as a primitive.

Definition 9.6 (Inertial delay function)
Let $ID := (t_{PLH}, t_{PHL}, t_0, i_1) \in \mathbb{N}_o^4$, $t_{PLH}+t_{PHL}+i_0+i_1$ finite, $i_0 < t_{PHL}, i_1 < t_{PLH}$ be an inertial delay specification. (i specifies how long a signal has to be stable to be accepted.)

mid $:= \max\{t_{PHL}, t_{PLH}\} + \max\{i_0, i_1\}$.

The function $idf_{ID} : FV^T \rightarrow FV^T$ is called inertial delay function with respect to ID

$:\Leftrightarrow$

$\exists(idf' : FV^{\text{mid}} \rightarrow FV) : \forall a \in FV^T : \forall t \in T :$

$(idf_{ID}(a))_t = idf'(a_t,\ldots,a_{t-\text{mid}})$ *with*

$$idf'(a_t,\ldots,a_{t-\text{mid}}) = \begin{cases} pr_{t-t_{PLH}}(a) & \text{if } a_t \in \{1,p\} \vee (a_t = u \wedge t_{PHL} \geq t_{PLH}) \\ pr_{t-t_{PHL}}(a) & \text{otherwise} \end{cases}$$

with (for $d \in \{t_{PLH}, t_{PHL}\}$) :

$$\begin{aligned} pr_{t-d}(a) = u \Leftrightarrow \quad & i) \quad a_{t-d} = u \\ \vee\ & ii) \quad a_{t-d} = 0 \wedge \exists t > t' > t-d > t" > t - mid : \\ & \qquad a_{t-d} \neq a_{t'} \wedge a_{t-d} \neq a_{t"} \wedge t' - t" \leq i_0 \\ \vee\ & iii) \quad a_{t-d} = 1 \wedge \exists t > t' > t-d > t" > t - mid : \\ & \qquad a_{t-d} \neq a_{t'} \wedge a_{t-d} \neq a_{t"} \wedge t' - t" \leq i_1 \\ = a_{t-d} \quad & \qquad \text{otherwise} \end{aligned}$$

Example. Consider the inertial delay specification $id1 = (3,4,3,2)$. Then, the following transformation can be observed from signal a to signal b by idf_{id1}:

```
a  =  00000p1n00000p11111n0p1111
b  =  ????00000pun00000p11111nup1111
```

Inertial delay is supported by VHDL as a primitive; however, no distinction between different values is made.

Edge-triggering needs a bit more consideration. In a discrete model the slope of a transition is represented by the period a value p or d can be observed. In the case of a period longer than a certain threshold, the slope is assumed to be too low for the edge to be accepted as trigger.

Definition 9.7 (Edge triggered inertial delay function)
Let $ED := (t_{PLH}, t_{PHL}, t_0, i_1, ep, en) \in \mathbb{N}_0^6, (t_{PLH}, t_{PHL}, i_0, i_1)$ be a valid inertial delay specification, $ep + en$ finite be an edge triggered inertial delay specification. (By ex the longest time-period is denoted during which a signal has the transition value x to be accepted as trigger, 0 stands for "no edge triggering").
$med := \max\{t_{PLH}, t_{PHL}\} + \max\{i_0, i_1, ep, en\}$

The function $edf_{ED} : FV^T \rightarrow FV^T$ is called <u>*edge triggered inertial delay function*</u>

$:\Leftrightarrow$

$$\exists (edf' : FV^{\text{med}} \rightarrow FV) : \forall a \in FV^T : \forall t \in T :$$
$$(edf(a))_t = edf'(a_t, \ldots, a_{t-\text{med}}) \textit{ with}$$

$$edf'(a_t, \ldots, a_{t-\text{med}}) = \begin{cases} pr_{t-t_{PLH}}(a) & \text{if } a_t \in \{1, p\} \vee (a_t = u \wedge t_{PHL} \geq t_{PLH}) \\ pr_{t-t_{PLH}}(a) & \text{otherwise} \end{cases}$$

with (for $d \in \{t_{PLH}, t_{PHL}\}$):

$$\begin{array}{rll}
pr_{t-d}(a) = u \Leftrightarrow & i) & a_{t-d} = u \\
& \vee\ ii) & a_{t-d} = 0 \wedge \exists t > t' > t-d > t" > t - med : \\
& & a_{t-d} \neq a_{t'} \wedge a_{t-d} \neq a_{t"} \wedge t - t" \leq i_0 \\
& \vee\ iii) & a_{t-d} = 1 \wedge \exists t > t' > t-d > t" > t - med : \\
& & a_{t-d} \neq a_{t'} \wedge a_{t-d} \neq a_{t"} \wedge t - t" \leq i_1 \\
& \vee\ iv) & ep \neq 0 \wedge a_{t-d} = p \wedge \exists t > t' > t-d > t" > t - med : \\
& & \forall t' \geq t^+ \geq t" : a_{t+} \in \{a_{t-d}, u\} \wedge t' - t" \geq ep \\
& \vee\ v) & en \neq 0 \wedge a_{t-d} = n \wedge \exists t > t' > t-d > t" > t - med : \\
& & \forall t' \geq t^+ \geq t" : a_{t+} \in \{a_{t-d}, u\} \wedge t' - t" \geq en \\
= 1 \Leftrightarrow & vi) & ep \neq 0 \wedge a_{t-d} = p \wedge \exists t > t-d > t' > t - med : \\
& & a_{t-d+1} = 1 \wedge a_{t'} = 0 \wedge t - d + 1 - t' < ep \\
& & \wedge pr_{t-d+1}(a) \neq u \wedge pr_{t'}(a) \neq u \\
& \vee\ vii) & en \neq 0 \wedge a_{t-d} = n \wedge \exists t > t-d > t' > t - med : \\
& & a_{t-d+1} = 0 \wedge a_{t'} = 1 \wedge t - d + 1 - t' < en \\
& & \wedge pr_{t-d+1}(a) \neq u \wedge pr_{t'}(a) \neq u \\
= 0 & & \text{otherwise}
\end{array}$$

Comments. By inertia the delayed value is mapped on u whenever it is not stable long enough (ii, iii). The edge-triggering maps a transition to value 1 at a single point of time at the end of the edge if the slope of the transition is "high enough" (vi, vii). Smooth edges are mapped on u, too (iv, v). All other cases are mapped to 0.

Example. Consider the edge triggered inertial delay specification $ed_1 = (4, 4, 3, 2, 2, 0)$. Then the following transformation occurs from signal a to signal b by edf_{ed1}:

```
a  =  00000p1n000000ppp11111nnn00000pp11111
b  =  ????000000u000000uuu00000000000000010000
```

As the last part of the assembly kit, transport delay functions are defined that effect the "slope" either by increasing or decreasing it.

Definition 9.8 (Smoothing transport delay function)
Let $SD := (t_{PLH}, t_{PHL}, su, sd) \in \mathbb{N}_0^2 \times \mathbb{Z}^2, t_{PLH} + t_{PHL} + |sv| + |sd|$ *finite,* $su < t_{PLH}, sd < t_{PHL}$ *be a smoothing transport delay specification. (The values* su *and* sd *are additive constants, with the meaning that a slope represented by a sequence of* n *symbols* x *is transformed to one represented by a sequence of* $n + m$ *symbols,* $m \in \{su, sd\}, su$ *affecting positive edges,* sd *negative ones.)*

$msd := \max\{t_{PLH}, t_{PHL}\} + \max\{su, sd\}$

$sdf : FV^T \rightarrow FV^T$ *is called* <u>*smoothing transport delay function*</u>

$:\Leftrightarrow$

$\exists (sdf' : FV^{\mathrm{msd}} \rightarrow FV) : \forall a \in FV^T : \forall t \in T :$

$$(sdf(a))_t = sdf'(a_t, \ldots, a_{t-msd}) \text{ with}$$

$$sdf'(a_t, \ldots, a_{t-msd}) = \begin{cases} pr_{t-t_{PLH}}(a) & \text{if } a_t \in \{1, p\} \vee (a_t = u \wedge t_{PHL} \geq t_{PLH}) \\ pr_{t-t_{PHL}}(a) & \text{otherwise} \end{cases}$$

with (for $d \in \{t_{PLH}, t_{PHL}\}$*):*

$$\begin{aligned}
pr_{t-d}(a) = 0 \Leftrightarrow &\quad i) \quad a_{t-d} = 0 \wedge sd \leq 0 \\
\vee &\quad ii) \quad a_{t-d} = 0 \wedge sd > 0 \wedge \forall t - d - sd \leq t' \leq t - d : a_{t'} \neq n \\
\vee &\quad iii) \quad a_{t-d} = n \wedge sd < 0 \wedge \exists t - d \leq t' \leq t - d - sd : \\
&\qquad\qquad a_{t'} = 0 \wedge a_{t-d-1} = n \\
= 1 \Leftrightarrow &\quad iv) \quad a_{t-d} = 1 \wedge su \leq 0 \\
\vee &\quad v) \quad a_{t-d} = 1 \wedge su > 0 \wedge \forall t - d - su \leq t' \leq t - d : a_{t'} \neq p \\
\vee &\quad vi) \quad a_{t-d} = p \wedge su < 0 \wedge \exists t - d \leq t' \leq t - d - su : \\
&\qquad\qquad a_{t'} = 1 \wedge a_{t-d-1} = p \\
= p \Leftrightarrow &\quad vii) \quad a_{t-d} = p \wedge su \geq 0 \\
\vee &\quad viii) \quad a_{t-d} = p \wedge su < 0 \wedge \forall t - d \leq t' \leq t - d - +su : a_{t'} \neq 1 \\
\vee &\quad ix) \quad a_{t-d} = p \wedge su > 0 \wedge \exists t - d \leq t' \leq t - d + su : a_{t'} = p \\
\vee &\quad x) \quad a_{t-d} = p \wedge su < 0 \wedge \exists t - d \leq t' \leq t - d - su : \\
&\qquad\qquad a_{t'} = 1 \wedge a_{t-d-1} \neq p \\
= n \Leftrightarrow &\quad xi) \quad a_{t-d} = n \wedge sd \geq 0 \\
\vee &\quad xii) \quad a_{t-d} = n \wedge sd < 0 \wedge \forall t - d \leq t' \leq t - d - +sd : a_{t'} \neq 0 \\
\vee &\quad xiii) \quad a_{t-d} = n \wedge sd > 0 \wedge \exists t - d \leq t' \leq t - d + sd : a_{t'} = n \\
\vee &\quad xiv) \quad a_{t-d} = n \wedge sd < 0 \wedge \exists t - d \leq t' \leq t - d - sd : \\
&\qquad\qquad a_{t'} = 0 \wedge a_{t-d-1} \neq n \\
= u &\qquad \text{otherwise}
\end{aligned}$$

Example. Consider the smoothing transport delay specification $sd_1 = (2, 2, -1, 1)$. Then, the following transformation exists from signal a to signal b by sdf_{sd1}:

```
a  =  0000ppp1111nnn0000pp1111nn00p11n00
b  =  ??0000pp11111nnnn000p11111nnn0p11nn0
```

In VHDL, neither edge triggered inertial delay functions nor smoothing transport delay functions are offered as primitives. They may, however, be programmed by a user. The four delay functions defined above may be replaced by other ones if different effects have to be modeled. In any case, they may serve as typical examples of such functions. Using such delay functions as a kit, quite realistic real time switching functions can be defined. As an example, this is achieved by composing a smoothing transport delay function, a timeless Boolean function and n edge triggered inertial delay functions. Other combinations are possible as well, but it seems to be the most natural approach to locate smoothing at the output of a switching element, while its inertia and edge sensing are associated with its inputs.

Definition 9.9 (Real time switching function)
Let $SD := (ot_{PLH}, ot_{PHL}, su, sd)$ be a smoothing transport delay specification, $ED_i := (it_{PLH}, it_{PHL}, t_0, i_1, ep, en)_i, i = 1 : n$ edge triggered inertial delay specifications, sdf_{SD} a smoothing transport delay function, edf_{ED_i} edge triggered inertial delay functions ($i = 1 : n$), and f a timeless Boolean function with n arguments.

$$\begin{aligned} mdi &:= \max_{1:n}\{it_{PLH_i}, it_{PHL_i}\}, mdo := \max\{ot_{PLH}, ot_{PHL}\}, \\ min &:= \max_{1:n}\{i_{0_i}, i_{1_i}, ep_i, en_i\}, ms := \max\{su, sd\}, \\ his &:= mdi + mdo + min + ms \end{aligned}$$

$rsf : ((FV)^T)^n \rightarrow FV^T$ *is called* <u>*real time switching function*</u>
$:\Leftrightarrow$
$\exists rsf' : ((FV)^n)^{his} \rightarrow FV : \forall a \in ((FV)^T)^n : \forall t \in T :$
$\quad (rsf(a))_t = rsf'(a_t, \ldots, a_{t-his})$ *with*

$$\begin{aligned} rsf'_t &= sdf_{SD}\ (f_t, \ldots, f_{t-mdo}) \\ &= sdf_{SD}\ (f(\ edf'_1(a1_t, \ldots, a1_{t-mdi-min}), \ldots, \\ &\qquad\qquad edf'_1(am_t, \ldots, am_{t-mdi-min}) \\ &\qquad\quad), \ldots, \\ &\qquad\quad f(\ edf'_1(a1_{t-mdo}, \ldots, a1_{t-his}), \ldots, \\ &\qquad\qquad edf'_m(am_{t-mdo}, \ldots, am_{t-his}) \\ &\qquad\quad) \\ &\qquad) \end{aligned}$$

The approach presented above allows for relative precise modeling of timing effects. It may easily be modified to support logics with another value-set, or to describe other timing effects.

The question arises, whether it makes sense to model the timing behavior in such a precise way. Definitely, it is not necessary at the algorithmic level and at register-transfer level. It can even be omitted at the logic level as long as strictly synchronous designs with a well calculated clocking rate are used. At the algorithmic level, in most cases a causal timing is used, i.e., only the relative ordering of actions is specified. For this purpose, algorithmic constructs or explicit causality structures using preconditions/postconditions are sufficient.

The typical timing model at the register-transfer level is the counting of clock ticks. All other actions are related to these clock ticks. This concept can be modeled very easily. In a strictly synchronous design this relation to clock ticks exists at the logic level as well. However, for performance reasons many designers tend to use less strict synchronization schemes. Hence, whenever interactive synthesis has to be supported by simulation, certain timing tricks used by a designer have to be considered. To handle these timing tricks in analytical methods is very difficult, though attempts are made to cover them, e.g., [SaBo91]. Therefore, time accurate modeling is a domain, in which simulation really can prove its strength. However, accurate timing costs a lot of simulation performance. Clever designers will, therefore, model only as accurately as really necessary, and clever implementors of simulation systems will design adaptive algorithms that model only as accurately as needed for the actual design being simulated.

In this section an abstract model of timing has been introduced. Hardware description languages are intended only to specify the timing parameters. In most cases, the point of view is different from the one used in this section. Timing has been introduced by defining current values as a function of past sequences of values (backward oriented modeling) while in most hardware description languages timing is expressed by scheduling future events (forward oriented modeling). Backward modeling is mathematically easier to describe. It allows for specifying timing simply by functions while forward oriented modeling needs a procedure of scheduling and, possibly, rescheduling of events. CONLAN [Pilo83] is one of the few hardware description languages completely bound to backward oriented modeling. VHDL, on the other hand is mostly forward oriented, but supports backward oriented modeling to a limited extend as well.

9.3 Simulation Techniques

The task of a simulation algorithm comprises the effort to map a modeling concept (or a variety of modeling concepts) onto the architecture of a host computer. Of course, the most efficient simulation is achieved if the architecture of the host computer is identical or at least similar to the modeling concept to be supported.

This is the basic idea of a class of dedicated simulation machines (hardware accelerators) [BDPV88, HaFi85, HaHE90, KoNT90]. Another class of such machines makes use of pipelining to accelerate sequential algorithms.

In most cases, however, a conventional von Neumann computer has to be used as host computer for simulation. This case is discussed in the following. The main problem of mapping onto a strictly sequential machine is the high degree of parallelism usually contained in concepts for hardware implementations.

There are three main techniques to tackle the problem:

- *Streamline Code Simulation* (SCS)
- *Equitemporal Iteration* (EI)
- *Critical Event Simulation* (CES)

These approaches are discussed in the following sections.

9.3.1 Streamline Code Simulation

The basic idea of this approach is to generate code from the hardware description that can be executed directly by the host computer. Therefore, this technique is often called *compiled mode simulation*. Of course, it is possible to generate directly executable codes for any modeling technique, but SCS in its strict sense, without any interpretative and scheduling components, usually is restricted to

- continuous evaluation as modeling concept,
- combinational or strictly synchronous circuits,
- models for which timing information is not important.

The classical application area of SCS is the simulation of combinational circuits at the logic level. This example, therefore, is presented first.

A combinational circuit can be represented as a *directed acyclic graph* (DAG). The nodes of the DAG represent the gates of the circuit, while each connection from a gate output to a gate input is represented by an edge of the DAG (cf. Figure 9.1).

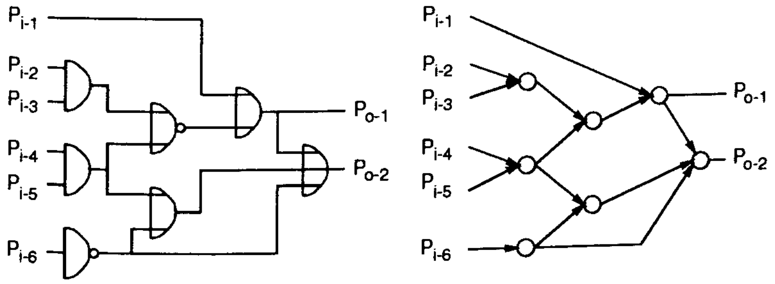

Figure 9.1: Combinational circuit and its representation as a DAG.

It is a straightforward task to semi-order the nodes of a DAG with respect to the length of the longest path from a node to a primary input. This technique is called *levelizing*:

1. Assign to each primary input in_i : $level(in_i) = 0$

2. Assign to each other node n_j : $level(n_j)$
   $= (1 + \max\{level(n_k) | \exists\ edge\ from\ n_k\ to\ n_j\})$

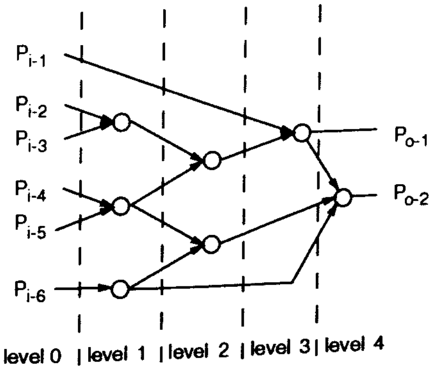

Figure 9.2: Levelized DAG from Figure 9.1.

Then, the levels may be interpreted as follows:

i) There is no influence of a node at level i on a node at level $j, j < i$.

ii) There is no cross-influence between nodes at the same level.

iii) A node at level i may (but is not required to) influence any node at level $k, k > i$.

By levelizing, therefore, a dependency relation is introduced for the circuit. This is a further abstraction of the dependency relation already given by the DAG. It is precise enough to define a save sequence of calculations at the different nodes, i.e., a sequence in which no values are referred to before they have been calculated: it is only necessary to arrange the code in accordance to ascending levels of the represented nodes. The sequence within a level is arbitrary because no interdependence between nodes of the same level exists.

In the case of logic-level simulation, the code needed for a gate is given by only a few instructions of the target computer. The connecting nets are represented as variables (memory locations in the main or virtual memory of the host computer). The example used in Figure 9.1 may be translated into the PASCAL-like code shown in Figure 9.3:

```
var pi_1, pi_2, pi_3, pi_4, pi_5, pi_6: word;
    int_1, int_2, int_3, int_4, int_5: word;
    po_1, po_2: word;
begin
 { level 1 computations }
 int_1 := pi_2 and pi_3;
 int_2 := pi_4 and pi_5;
 int_3 := not pi_6;
 { level 2 computations }
 int_4 := int_1 nor int_2;
 int_5 := int_2 nor int_3;
 { level 3 computations }
 po_1  := pi_1 or int_4;
 { level 4 computations }
 po_2  := po_1 or int_5 or int_3;
end
```

Figure 9.3: Code for example used in Figure 9.1.

The PASCAL-like language used as the target code in this example can easily be replaced by the machine code of an arbitrary processor, replacing declarations by allocating memory locations and assignment statements by executable instructions of the target machine. The example above describes a calculation based on one single input pattern. It may easily be extended to process an arbitrary sequence of input patterns. By introducing simulated shift registers arbitrary sequential circuits can be simulated as well, even asynchronous ones. The latter is of minor interest in the context of synthesis as long as no tricky manual modifications are performed in an interactive synthesis environment. Usually, when SCS is applied, sequential circuits are supported using another and much simpler (but less accurate) approach: It is assumed that no timing information of granularity finer than clock ticks is required. Thus, any sequential circuit may be represented in Huffman normal form.

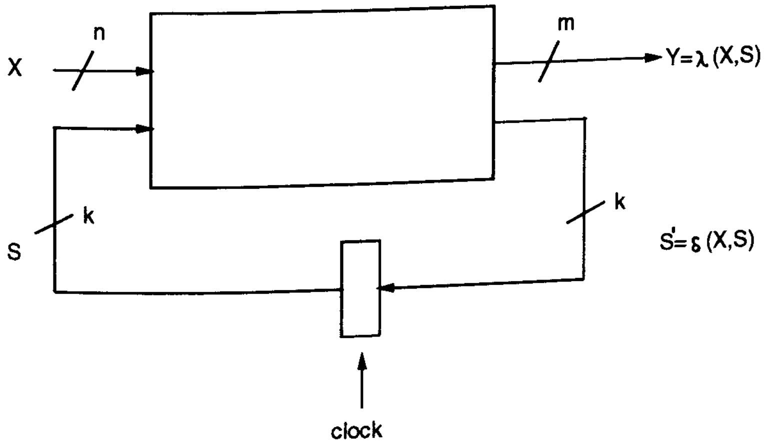

Figure 9.4: Sequential circuit in Huffman normal form.

Two combinational functions have to be calculated:

- $y = \lambda(x, s)$ (Mealy model)
- $s' = \delta(x, s)$.

These functions can be calculated (in a timeless manner) by the basic SCS approach. If one loop iteration of the algorithm is interpreted as one clock cycle, state register on the feedback loop of the circuit do not need to be modeled explicitly. It is sufficient that the assignment to the variables representing the state takes place at the end of each iteration.

SCS is a very efficient approach. It requires more effort in the compilation phase but avoids any overhead caused by interpretation. Therefore, whenever applicable, compiled mode simulation becomes more and more the preferred method. In the context of synthesis, especially high performance register-transfer level simulation can be achieved using this method. The approach presented above to support precise timing, however, causes an enormous overhead. Whenever such precise models are requested (and simulation seems the technique best suited in this area) other techniques are preferable. They are outlined below.

9.3.2 Equitemporal Iteration

As already mentioned, especially those levels of abstraction that are used as input levels to synthesis need to be supported by simulation. For high-level synthesis this is the algorithmic level. SCS is a general technique, but not necessarily the most efficient one at the algorithmic level. A classical, very simple table-driven simulation is *equitemporal iteration* (EI). It is directly applicable to VHDL. As for SCS, an iterative approach is used in EI to calculate the state of the entire system being simulated. After each iteration the global simulated time is increased by a fixed increment. The step size may vary from iteration to iteration, but is always equal for all components of the system to be simulated. (In this context, this technique is referred to as equitemporal.)

The EI-algorithm is very simple and easy to implement. Unfortunately, it is rather inefficient if the percentage of components being executable at a certain point of time is low. This is due to the fact that at every point of time (within the observation resolution) each component has to be visited and checked whether its executability condition is true. At the logic level, typically, the probability for this is about 0.05. The relative efficiency of EI increases with the instability ratio of the system to be simulated. This instability ratio increases either if the resolution of observation time is decreased, or if a larger domain of observable values is used. Consequently, EI is mostly used at the RT-level (coarse time resolution) and the circuit level (continuous domain of observable values).

9.3.3 Critical Event Scheduling

This is the most frequently used simulation technique. The approach is aimed at the task to overcome the efficiency problems of EI by restricting calculations to necessary (i.e., non identity) calculations. CES can be applied to any modeling concept in which the following restrictions are respected:

i) The point of time of the next event is predictable.

ii) If the point of time at which a certain event takes place next, is not predictable, this event does not take place, unless it becomes predictable by another event taking place.

These restrictions are fulfilled by all modeling concepts to be considered in our context, thus leaving CES to be a universal algorithm in the field of synthesis-related simulation.

For CES it is assumed that the system to be simulated is partitioned into components at compile time. Furthermore, a dependency relation has to be introduced. If there is a direct dependency of component A from component B, A is called *influencer* of B and B *influencee* of A. This dependency relation allows the CES-algorithm for the decision which components are influenced by the assignment of a new value to a data object. If the necessary timing information is provided, in CES-algorithm it can also be decided precisely when this has to happen. Only those parts of the system being influenced by an event, and only those points of time an event is scheduled for, need to be considered further. Therefore, in contrary to EI, CES is a local approach.

In Figure 9.5 a skeleton of the algorithm is shown:

```
type event = record component_id, event_time: integer;
                    new_value: word
             end;
var current_event, new_event: event; event_queue: queue of event;
begin
  simulated_time := 0;
  while simulated_time <= final_time & queue_level <> empty do
  begin
    current_event := remove (event_queue);
    current_time := current_event.event_time;
    changed := data [current_event.component_id] <>
                    current_event.new_value;
    if changed then
    begin
      data[current_event.component_id] := current_event.new_value;
      if no_of_influencees[current_event.component_id] <> 0
      then
        for i:= 1 to no_of_influencees[current_event.component_id]
        do
        begin
          component:=
             influencee [current_event.component_id, i];
          if executable (component) then
          begin
            new_event.component_id := component;
            new_event.new_value := action (component);
            new_event.event_time := current_time+
                                    elapses (component);
            insert (event_queue, new_event)
          end
        end {for loop body}
      queue_level := test_level (event_queue)
    end {if changed then begin}
  end {while loop body}
end
```

Figure 9.5: Skeleton of the CES algorithm.

Some comments. In the algorithm shown in Figure 9.5 the existence of an event queue which is always kept sorted in ascending order with respect to the event times is assumed. The operation *remove* deletes the first element from the queue, the operation *insert* inserts an element into the queue, preserving the order of elements, and, finally, the operation *test_level* returns the current number of elements in the queue. The algorithm first sets the initial value of the simulated time (line 6) and, then, starts the main loop (lines 7–34). The algorithm is stopped if either the final time is reached or no more events are left (line 7). The loop body starts by extracting the first element of the event queue (lines 9, 10) and checking whether the new value to be assigned differs from the old one (line 11). The special treatment of this question is called *selective trace*. It can be observed that time is not incremented in fixed steps; rather, the simulated time is changed directly to the point of time of the extracted event. If a change of value occurs and influencees exist, all these influencees have to be processed (lines 13 and 17). If the influencee is executable, an event has to be created and inserted into the queue (lines 24–30).

This algorithm has to be combined with a data structure which describes the structure of the system to be simulated; to generate it, the description of the system is compiled. It provides storage elements for all data objects to be considered (assumed to be arranged in a one-dimensional array, see lines 11 and 15), information on the dependency relation (again assumed to be arranged as arrays, see lines 16, 18, 22), component identification (line 25), component operation (line 26), and component delay time (line 28). The clever organization of these data structures heavily influences the performance of the simulator.

The CES algorithm, too, is relatively simple. It is very efficient in most cases, because no time is wasted for unnecessary calculations. This has to be paid for by the requirement to keep the event queue sorted at all times. Insertions into and deletions from priority queues can be accomplished in $O(\log n)$ time, with n being the number of items in the queue. This is acceptable, especially if one considers that the event queue is reasonably short at any time. This is due to the fact that at most times approximately 95% of a circuit is stable, i.e., only few future events are known.

In situations where maintaining the priority queue tends to become the bottleneck of the implementation, the time-wheel approach can be used. The key idea of this approach is to use a circular data structure with a fixed number n of slots to implement the priority queue. Each slot i contains all known future events to take place at the simulated point of time

$$time = (starttime\ of\ actual\ cycle\ + i) \bmod n,$$

i.e., events to be scheduled are inserted into the proper slot. This is nothing else than a special implementation of a priority queue (based on the address calculation method used in the well-known bucket sort), which is possible whenever the priorities can be mapped to discrete values. If the number of slots is a power of two, the identification of the proper slot can be realized by a simple masking operation.

In the CES-algorithm, the simulated time advances in a cyclic manner by a fixed increment stepping forward to the next slot of the time wheel. Then it is checked whether an event to be executed is contained in this slot. If so, it is processed as described in the above algorithm. If not, the next slot is inspected, etc. The fixed time increment is due to the similarity between CES with time-wheel and EI. However, in contrast to EI, not the entire circuit is inspected at each point of time, but only the few active parts.

CES as a very general and efficient algorithm is ideally suited for multi-level/mixed-level simulation, since it supports the algorithmic level equally well as the register-transfer and logic level. Because mixed-level simulation systems are best suited for synthesis, CES seems to be the adequate implementation technique. In addition, it is an ideal algorithm for the entire bandwidth of VHDL.

9.4 Multi-Level Simulation

As already mentioned, within the context of synthesis, simulators are needed that are suited for different levels of abstraction, e.g., to experiment with the specification of a system or to verify one possible implementation. Classical simulators dedicated to a specific abstraction level are not useful for this purpose. Simulation systems are needed that support at least two different levels of abstraction. In addition, any support for comparison of simulation results obtained at different levels of abstraction, applying equivalent stimuli, is helpful.

The term ***multi-level simulation*** refers to a simulation system that covers more than one level of abstraction. Since by this definition a multi-level simulation system is not necessarily capable of supporting simulations at various levels *concurrently*, often the more precise term ***mixed-level simulation*** is used. Two main approaches to multi-level/mixed-level simulation can be distinguished:

- either a ***broadband simulator*** is offered
- or a set of dedicated simulators is integrated reasonably well into one system (***multi-simulator***).

9.4.1 Broadband Simulator

A language like VHDL requires a flexible simulation system covering various levels of abstraction and various styles of description. A single, monolithic simulator seems to be the most natural approach. The system to be simulated can be treated, as described earlier, without the application of any transformation and without having to deal with distributing simulation tasks to various simulators. No tricky modifications of descriptions in order to overcome restrictions of special simulators are necessary. It can be seen that at least based on CES an algorithm powerful enough to support the entire bandwidth of VHDL can be implemented.

Of course, one has to pay for this advantage. Due to the bandwidth covered by VHDL, a complex simulator has to be implemented, which has to support various modeling concepts and heterogeneous domains. Often, very general algorithms, when applied to special cases, tend to be rather inefficient compared to dedicated solutions. The reason is that dedicated algorithms incorporate the knowledge about specific restrictions while general solutions have to cover and to check for a variety of cases, which do not occur in a specific situation. In the case of VHDL, the performance can be increased by translating large parts of a description, especially the sequential bodies of processes, into directly executable code. Then, only the basic handling of concurrency and timing is performed by an interpretative CES-algorithm of high performance. High performance can be achieved because of its restriction to this special type of events. Today's compiler generation systems support the implementation of this approach.

9.4.2 Multi-Simulators

Despite of the advantages mentioned for broadband simulators, multi-simulators constitute reasonable solutions, as well. A typical situation, in which multi-simulators are useful, arises if a semiconductor manufacturer accepts only models suitable for the proprietary simulator. Another problem is the lack of availability of models, either of standard components or of proprietary parts. Often, a large library exists which was written for a specific simulator. The most important application of multi-simulators is the simulation of a completely heterogeneous design specification. Mixed digital/analog simulation provides a typical example. Systems comprising a mixture of electronic and mechanical parts (mechatronic systems) are of similar nature; such systems begin to play an increasingly important role.

Three major problems need to be solved if existing dedicated simulators have to be coupled:

i) the data exchange between the various simulators,

ii) the synchronization of the simulators involved,

iii) a user interface as transparent as possible with respect to the heterogeneity, i.e., hiding the different simulation approaches from the user.

Compared to the other three problems, data exchange [BeLW91] seems easier to solve. Data have to be exchanged on global connections. These connections are those residing in the intersection of the set of primary inputs of a specific model and the primary output of another. In the ideal (but rare) case, the signal value domains of the primary inputs and outputs involved are identical. Then, no conversion of signal values is necessary at all.

Some other cases are relatively simple to handle: those, where a larger domain of values can be partitioned into classes such that each class can be bijectively mapped into a single value of a smaller domain. In any case, it is always reasonably simple to map a larger domain onto a smaller one, a problem typically to be solved when signals are sent from a model at a lower level of abstraction to one at a higher level of abstraction. The opposite direction is more difficult to handle. In this case, assumptions on details of the driver circuit of the sending component are necessary. These assumptions can be made either globally or locally, based on user specifications. Mapping digital signals to inputs of analog simulators is a typical example for a non-trivial problem of this class.

Synchronization is the central problem of multi-simulation. Since the entire system of cooperating simulators models the system to be simulated, the simulators involved have to be kept at least synchronous enough so that at each point of time when a data exchange happens, this point of time is legal for all simulators involved. This definition gives some degree of freedom concerning the synchronization method used. There are two main approaches:

- the *supervisor approach* and
- the *time-warp method*.

The supervisor approach is an over-synchronizing "pessimistic" solution. It requires each simulator involved to hand over control to the supervisor before the simulator advances its modeled time. The supervisor is then able to identify the simulator that plans to schedule an event in the near future; this simulator is activated by the supervisor. The approach is flexible enough to handle all kinds of simulators. To integrate a CES simulator, this algorithm may contact the supervisor just after having extracted the next event from the event queue. Simulators of class SCS or EI may contact the supervisor just before initiating a new cycle. The supervisor approach is very simple, easy to implement, and requires only minor modifications of the simulators involved. Unfortunately, it over-synchronizes the system as no overtaking between simulators is allowed even if no communication occurs during the overlap period. The most serious drawback is the fact that only one simulator can be active at a time. Therefore, it is rarely applicable to multiprocessor systems.

While the supervisor approach over-synchronizes, the time warping method [JeSo82] under-synchronizes. In this method, following a purely optimistic approach, completely free running simulators are assumed without any synchronization besides the exchange of messages. When such messages arrive at a point of time within the local past of a simulator, they cause this simulator to perform *roll-back*, a costly operation. This situation is discussed in more detail.

Let S_1 and S_2 be two simulators, currently simulating at local virtual points of time (LVT) t_{S1} and t_{S2}, $t_{S1} \neq t_{S2}$. It is assumed that S_1 sends a message m = (data,t_m) to S_2 where *data* denotes the transmitted value and t_m the point of time it has to come into effect. Of course, in any case the following holds:

i) $t_m \geq t_{S1}$.

Concerning t_{S2} there are two possibilities:

ii) $t_m > t_{S2}$ or

iii) $t_m \leq t_{S2}$.

Case ii) is simple. The message is handled like any stimuli event, i.e., it is inserted into the event queue.

Case iii) is much more complex. In this situation, S_2 has to be rolled back to a point of time t_h prior to t_m. This ability is required for any simulator to be integrated into a time-warp controlled multi-simulator system. While S_2 is resimulated starting from t_h, all external events between t_h and t_{S2} have to be reconsidered. Therefore, simulators to be integrated need to store the entire history of incoming events (at least for a period determined by the allowed slack between the simulators involved). The handling of past events is accomplished in three phases. In the first phase, called *roll-back*, the state of S_2 at point of time t_h is restored. S_2 may have sent messages to other simulators during the period between t_h and t_{S2}. All these messages, then, have to be canceled. This is performed during a *cancellation phase* by sending an *anti-message* for each message. These anti-messages are handled similarly to ordinary messages. If a simulator receives an anti-message that exists in its local future, this message is removed only from the event queue. If it is located in the local past, it causes a roll-back as well, with the respective message being removed from the stored incoming events. A single roll-back, therefore, may "ripple" through the simulation system by implicitly causing additional roll-backs. Finally, S_2 can resume its normal mode of operation.

The time-warp method, first introduced in [JeSo82], is a very general method and can be implemented efficiently on multiprocessor systems [Apos89, BaSK91, Jeff85a]. A further enhancement to this method has been achieved by *lazy cancellation* [Jeff85b]. Because the time-warp approach becomes less efficient if too many roll-backs occur, it has rarely been used in heterogeneous simulation systems. An example of a heterogeneous simulation framework for integration of arbitrary simulators is the SiCS system [BeLW91, Niem91]. This framework supports both the supervisor approach and the time-warp mechanism with a controllable slack.

To build a unified user interface that introduces transparency to the heterogeneous structure is the last problem to be solved by a multi-simulator system. Such an interface has to allow for the description of circuits in the proprietary language (including graphical ones) of the individual simulators, but should offer a unified description language (e.g., VHDL) as well. The same is true for stimulation and presentation of results. The entire control of the system has to be presented to the user in a strictly unified way. Embedding a multi-level simulation system into an electronic design automation framework [KlKu91, LeWB91, RaWa91] seems to be the most appropriate approach for this purpose.

Multi-level/mixed-level simulation as required in the context of synthesis is widely understood and available. Both monolithic broadband simulators and multi-simulator systems have their specific benefits. Multi-simulators have been studied for a long time [Merm85]. Recently, open frameworks for simulator coupling have been announced [Niem91].

9.5 Outlook

Only the kernel of a simulation system is presented in the preceding sections. The intent of simulation is to have a host system (a computer) show the same behavior as the system to be simulated, and then to perform useful experiments on this simulated system. Therefore, the quality of simulation is determined by

i) the quality of the model of the system to be simulated,

ii) the quality of the mapping onto the host system (i.e., the quality of the simulator itself),

iii) the quality of the experiment performed,

iv) the quality of result analysis.

Unfortunately, for the traditional simulation systems emphasis has been put mainly on i) and ii). The planning, executing and analysis of experiments have remained widely manual tasks.

In the simplest case, at least the application of stimuli and a certain post-processing of result patterns have to be performed. In graphics-oriented environments a graphical waveform editor is often offered for both purposes. This approach seems to be adequate at the logic level. It is convenient to use and it looks familiar to traditional design engineers. In the context of synthesis, this solution is less adequate. First of all, a synthesis-related simulation system has to cover higher levels of abstraction, levels at which waveforms are no longer a natural way of representation. More abstract stimuli have to be applied (e.g., machine programs for a processor to be synthesized) and much more abstract responses are expected (e.g., results of such programs). If low-level waveforms are of interest, they should be hierarchical so that certain patterns can be identified as the equivalent of abstract traces at the higher level of abstraction.

The most serious drawback of waveform editors is that they produce passive stimuli and passively record responses. In most cases, however, the system to be synthesized and, therefore, the system to be simulated as well, are embedded in a predefined environment. This environment, in general, reacts to outputs of the part to be designed, and as part of the reaction produces specific stimuli. Therefore, it is much more adequate to use a model of the environment instead of passive stimuli generators. Such a model can be written in the same language as that the system to be simulated is described in. Therefore, it can be executed by the same simulator which is used for the system to be exercised. This, of course, is true only if the hardware description language used is powerful enough. Because broadband simulation is assumed, this is the case. In the case of a multi-simulator system at least one simulator in the compound should be powerful enough to model even a complex environment.

VHDL uses exactly this approach of modeling the environment in the same language. This idea is perfectly supported by the modular structure of the language. With the aid of this language feature, it can also be expressed easily which part of a description has to be interpreted as a specification for synthesis and which part models only the environment for simulation purposes.

Replacing passive stimuli by a potentially interactive model of the environment does not at all exclude passive stimuli. They constitute the special case of a predetermined environment without any reaction to the simulated system. Such a special case of an environment can be described by the same language, possibly restricted to a subset.

The modeled environment interacting with the simulated system is the first step towards an environment in which experiments can be performed intelligently that help to answer specific questions about specific features of the system under simulation. If this is achieved, the modeled environment becomes a modeled experimentor that in a goal-oriented manner performs experiments directed by intermediate results and abstract features, extracted from these results. As performing such experiments is a non-trivial task which involves a lot of knowledge in order to be performed properly, a knowledge-based approach seems to be an adequate technique to tackle the problem.

Before describing a potential architecture of such a knowledge-based experimentor, the types of features an experimentor is looking for have to be examined. The easiest form of abstractions from values produced by a simulator are *virtual signals*. They are calculated by an arbitrary function using as arguments signal values or streams of such values, as they are produced by the simulator. Any kind of statistics may serve as an example for these virtual signals. A profound examination of simulation results alone, though only a part of an intelligent experimentor, can become a very complex task to be supported by sophisticated systems such as SIMUEVA [BuLa87, Busc90b]. In this section, a distinction is made between *validation, error correction, performance analysis, tolerance analysis, optimization*, and *stimuli validation*. Validation deals with questions, such as respecting predefined domains, delay ranges, bus restrictions, etc. In the case of error correction, hints are given as to what may be the reason for a detected error condition. Such evaluations are essential if a system like SIMUEVA is integrated into an entire experimentor, because for performance analysis delay paths, slopes of transitions, reaction times, etc., are calculated dynamically from simulation results. Tolerance analysis is similar to validation, but under certain assumptions it is calculated whether or not a proper operation is guaranteed over a range of possible operation conditions. Virtual signals for optimization purposes may include the dynamic power consumption, response times, distribution function of delays, etc. Finally, the applied stimuli can be analyzed with respect to the results obtained. Statistics about algorithmic paths visited during a simulation, coverage of cases at switches, grade of concurrency achieved, etc., are typical data of interest, in this case. These are questions similar to those of interest in software testing. This is not surprising, because validation of an initial specification for synthesis is a task similar to validating a piece of software by testing.

If a powerful result analysis system is available, the remaining step towards an intelligent experimentor can be attacked. This leads to the application of artificial intelligence techniques as some authors have indicated [AdPo86, ElÖZ86, EgRo88, LuAd86, Pfaf91, ShMA85]. An experimentor, in this case, becomes a special expert system using simulation results and possible additional external facts as basic facts, planned experiments as rules. By an inference mechanism, conclusions for controlling the execution of the experiment can be deduced. Using a simulation system with a hardware description language of sufficient power, the expert system itself can be built using this simulation system. Such an approach using DACAPO III [DACA90] as simulation system and hardware description language has been described in [Pfaf91]. Obviously, any hardware description language that allows a reactive description (e.g., VHDL) is suited to describe rules and, therefore, facts as well. The main question arises about the inference mechanism. But if one looks at the RETE algorithm used in OPS5 [Forg81], a similarity to CES with selective trace can be observed. Based on this similarity, a CES based inference mechanism for applications in the area of simulation has been described in [Pfaf90]. It depends on the services offered by the underlying simulation system whether the explanation component usually included in an expert system can be offered as well. In any case, it can be constructed by generating proper virtual signals because all information needed for producing explanations are available from the set of simulation results.

In the system described in [Pfaf91] arbitrary rules can be specified by a user. Thus, specific needs arising in a synthesis context can be supported. This system is built on a relatively complete set of basic rules. This rule base contains:

i) general rules such as value checking, cycle checking, etc.;

ii) analog rules such as transition slope checking;

iii) timing rules such as set-up and hold time checking, minimal pulse width checking, etc.;

iv) breakpoint rules affecting the operation of the controlled simulator;

v) tester-oriented rules, checking whether experiments can be performed by a hardware tester;

vi) abstraction level comparison to compare descriptions at various levels of abstraction;

vii) filtering rules for value sampling;

viii) rules to generate virtual signals such as error-functions, ratios, means, etc.;

ix) logic rules;

x) arithmetic rules.

This basic set of rules already allows formulation and performance of reasonably complex experiments that are used to guide a simulation run in a goal-directed way.

Knowledge-based experimentors provide a good argument for multi-simulator systems. If an experimentor is built on top of a powerful simulator, it can be plugged in very easily into any multi-simulator system; an ordinary simulator cooperating with a simulator-based experimentor automatically forms a multi-simulator system.

It can be expected that emphasis on experimentor guided simulation systems will increase. Then, the gap between analytical (formal) verification methods and simulation will become narrower. The main difference, then, will be that for the inference system of an analytical method (its theorem prover) static facts including symbolic signal values are used while a knowledge-based simulation system continues to be based on dynamic facts (simulation results). The emerging similarity—disregarding this difference—offers the opportunity to build cooperative systems including both approaches and inheriting the strengths of both methods. When tightly coupled with a synthesis system within a design framework [RaWa91] the ideal design environment will be achieved.

9.6 Problems for the Reader

1. Model the "philosophers problem" with CSP: Five philosophers are sitting around a table, each one looping between thinking and eating. For eating, a philosopher needs two forks but there are only five forks available, i.e., each pair of philosophers has to share the fork between them. Model the most simple solution, ignoring any deadlock problems.

2. Specify the operations *insert* and *remove* of the time wheel approach in PASCAL-like pseudo-code. The operation *insert* adds one event to slot *i* of the event-queue, *remove* removes the next event stored in the time wheel. Assume that the time wheel is modeled by a data structure of

```
type time_wheel_queue = array [0:2**size] of
                        record
                          component_id: integer;
                          new_value: word;
                          event_time: integer
                        end;
```