This is to certify that the

dissertation entitled

DESIGN METHODOLOGY IN HIGH LEVEL SYNTHESIS

presented by

Sea Hawon Choi

has been accepted towards fulfillment
of the requirements for

Doctor of Philosophy degree in Dept. of Computer Science

Dr. Moon-Jung Chung
Major professor

Date Nov. 15, 1995

Design Methodology in High Level Synthesis

By

Sea H. Choi

A Dissertation

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science

1995

Abstract

Design Methodology in High Level Synthesis

By

Sea H. Choi

As the complexity of hardware components and the tools involved increases, design process management becomes the key issue in improving productivity. Modeling the hardware system to be designed is the first, and one of the most important, steps of a design process for high level synthesis. In this thesis, new modeling techniques and a novel design process management scheme for high level synthesis, which help to increase the productivity and the quality of design, are presented.

The issues involved in the modeling of hardware systems and their integration with a hardware description language (HDL), especially the Very High Speed Integrated Circuit (VHSIC) HDL (VHDL), are addressed. First, several new techniques for modeling Finite State Machines (FSMs) using VHDL are presented. Then, these techniques are extended to parallel machines. It is shown that VHDL is an effective way of modeling a wide range of systems from a simple FSM to the structure and behavior of a parallel system. How different modeling techniques affect the quality of design is also demonstrated.

This thesis also proposes an execution environment which efficiently handles the design process. The environment evaluates design methodologies and assists designers in selecting and executing appropriate tools and methodologies. It also supports concurrent exploration of multiple design alternatives in a distributed environment. A salient feature of the proposed execution environment is that the execution environment is separated from the specification of methodologies. The proposed execution environment has been modeled using an Colored Petri-Net model. Both the implementation of the environment based on this modeling and benchmark results are presented.

To my wife and children

ACKNOWLEDGMENTS

First of all, I would like to give my sincere thanks to my wife, Juhee Choi, and two children, Esther and Eliot Choi, for their patience, endurance, and everlasting support. I also thank my parents and parents-in-law.

I wish to thank all the individuals who have assisted me in some way during my years of graduate study. In particular, the professional advice of my thesis advisor, Dr. Moon J. Chung, has been very beneficial to my development as a researcher and has helped to sharpen my focus on the many issues involved in hardware modeling and framework.

Table of Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Introduction

The increasing complexity of system design and the emergence of new technologies make the design process the key issue in the micro-electronics and computer-aided design (CAD) industries [1]. A complex system design has the following characteristics: hierarchical design, multiple design representations, and a large design space. A large system design typically includes multiple boards, a variety of implementation technologies, and interfaces. Design engineers must deal with many issues, such as partitioning among different boards, interface timing, and packaging.

There are three important issues in system design. The first issue is *design modeling.* Design modeling defines the functionalities of the design and verifies its correctness by simulation. These models not only ensure design correctness but also greatly affect synthesis results. *Design methodology or workflow management* in a Computer-Aided Design Framework must support a seamless way of carrying out the design process as well as a suitable way of representing the design process. It must also support tool encapsulation in order to carry out the design process. *Design*

data management deals with storing design data and capturing relationships such as version control and configuration binding. Design modeling and methodology management in relation to a novel execution environment and framework developed by this author (see description below) comprise the main issues discussed in this thesis. For a detailed analysis of design data management, see Kim [2].

This thesis presents issues involved in the design process in high level synthesis. First, design modeling techniques in the design process are discussed. Next, methodology management is discussed. Issues relating design modeling and methodology management involve the modeling techniques of systems to be designed and their integration with Very High Speed Integrated Circuit (VHSIC) HDL (VHDL). This thesis emphasizes the system modeling experiences and problems using VHDL (Part II). Different modeling techniques affecting the design process and the quality of design will also be shown. Models written in VHDL will be used as input descriptions to a proposed CAD framework (Part III).

In the relationship proposed by this thesis between modeling and design methodology management, modeling the prospective hardware system is the first, and one of the most important, steps of the design process. The relationship between design process and design hierarchies is represented by the *Y-Chart* [3, 4] shown in Figure 1.1. The design process can be formulated as a methodology traversing through the design space in a spiral fashion. More detailed information is added when we traverse toward the center. Typically, complex systems are designed in a hierarchical fashion using a top-down approach. Systems are modeled by building virtual prototypes. Desired modules can be gradually replaced by hardware components until the

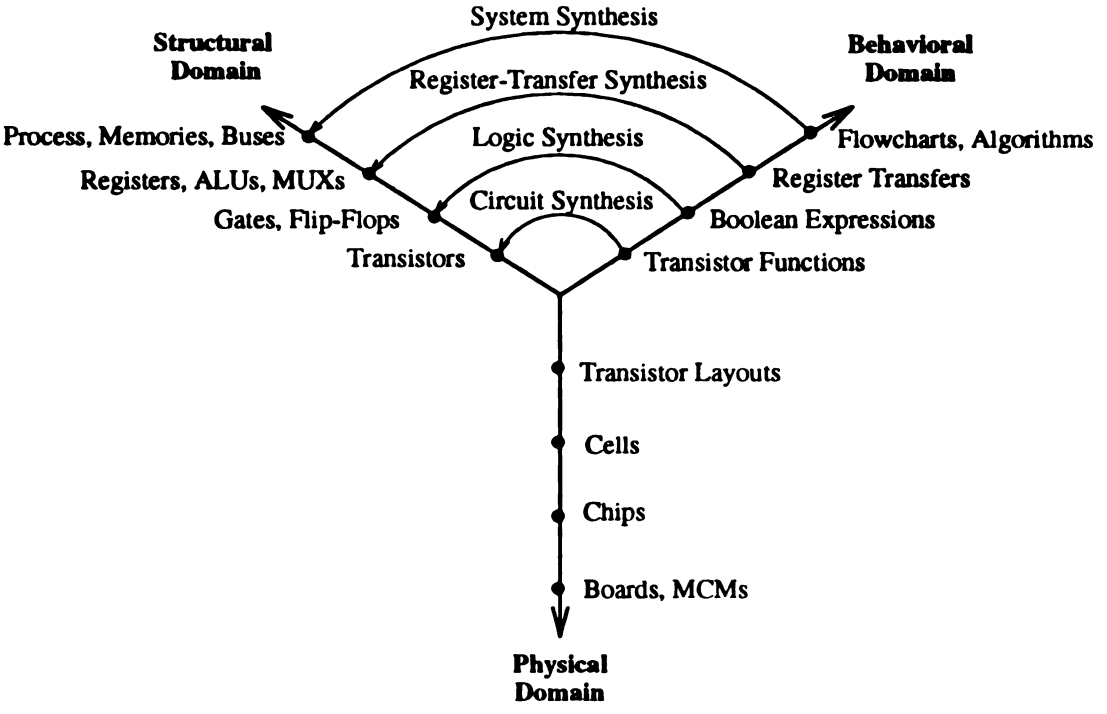


Figure 1.1: Y-Chart

final hardware is obtained. A top-down, hardware description language (HDL) based design methodology reduces the overall design time by virtual prototyping, verification, and synthesis. The pre-existing HDL model can be reused in the later design of components.

This thesis has selected an HDL to model hardware systems based on the effect of the particular language on design quality and time. VHDL was chosen because of its wide use in modeling the micro-electronic systems at the behavioral and structural levels. The virtual prototypes based on VHDL can be used in software components of hardware/software co-design environments. The software model can be verified using a software simulation environment or a hardware/software co-simulation environment. As this thesis discusses later, standard modeling styles and guidelines are crucial issues in concurrent design and integration as well as in hardware/software co-design and

integration.

This thesis illustrates the effectiveness of VHDL in modeling a wide range of systems, from a simple finite state machine (FSM) to complex parallel systems. Using the FSM, one of the most ubiquitous, and therefore important, among micro-electronic control devices, several new techniques of modeling FSMs using VHDL are presented. These techniques are then used to model the architectures of parallel machines and applications. The efficacy of these techniques is illustrated by implementing a real time signal processing application on a Splash 2 computer system. The same basic technique is used to model and synthesize an application, an on-the-fly sorting algorithm, in a hardware system. This thesis shows that different synthesis results are obtained from each different modeling approach and stresses the importance of modeling style.

As the complexity of computer systems increases, not only the modeling of such systems but also design methodology management take on critical importance. Effective design methodology management must support the following functions: specification and execution. Specification involves representing available methodologies and encapsulating them so that they can be searched and used. Execution identifies an appropriate methodology for a given task: this involves decomposing tasks hierarchically, selecting tools and invoking them, scheduling tasks with regard to hardware resources and available data, handling parallel exploration of alternatives, and backtracking.

A CAD framework is defined as "a software infrastructure that provides a common operating environment for CAD tools [5]." In order to meet the criteria of effective

design methodology management, the ideal CAD framework would provide a flexible, reconfigurable, reusable, and controllable design environment to designers. First, the separation of design methodology specification and execution should provide *flexibility* to the framework. Second, *controllability* of the ideal framework would allow the designer to intervene in the automatic execution of the existing framework. Third, the execution model of the well-designed framework should allow *formality* in the expression of the model itself.

Existing CAD Frameworks, such as ADAM [6, 7, 8], OCT [9], and Monitor [10] have several shortcomings. Flexibility in these frameworks is lacking since the specification of design methodology and execution are not separate and methodologies are fixed. Secondly, incorporating *process metrics* in methodology selection is difficult, representing dependencies between design data and methodologies is impossible, and systematic search of design space is problematic, all of which reduce controllability. Finally, since these approaches lack formalism, representing and storing the selected design process for reuse are not effective. These issues and their effect on high level synthesis will be addressed later in this thesis.

The CAD framework proposed in this thesis provides unique solutions to the problematic aspects inherent in other CAD frameworks. First, the proposed framework is highly flexible since it incorporates the separation of design methodology specification and execution. Second, the execution model of the environment is based on a Colored Petri-Net model [11, 12, 13], providing formality. Finally, this framework achieves a high degree of controllability through feasibility of intervention, backtracking, and simultaneous execution. Thus, this thesis presents a new *execution environment* for

managing the design process in high level synthesis which is able to support multiple functions.

The thesis is organized as follows: In Part II, various modeling techniques and examples are presented, showing how VHDL descriptions can be used in the design process as input files. In Part III, the proposed CAD framework execution environment is explained in detail. Related work and synthesis examples employing the CAD framework are also presented. Finally, conclusions drawn from the research are summarized in Part IV.

Part II

Modeling

Modeling the micro-electronic system is the important first step in the design process. Proper modeling technique is essential for rapid prototype and hardware/software co-design. Properly modeled components facilitate design reusability. Efficiently developed models can be used in the rapid prototyping of a system to be designed and can be reused in hardware synthesis. The resulting hardware model is not required to be synthesizable, although this is desirable. The hardware model can be implemented quickly and inexpensively using an HDL such as VHDL [14, 15, 16] or Verilog [17, 18]. The entire system model, both hardware and software, can now be realistically simulated and tested.

An HDL is necessary for a designer to describe the functionality of micro-electronic hardware parts. At each stage of the design, rapid prototyping using HDL can be used to explore design options and ensure that specifications meet design functionality requirements. Functional verification is usually done by simulation of the HDL specification. Initially, the prototype is entirely software, but as the design progresses, the prototype includes hardware for more and more subsystems until the final implementation is developed. The HDL modeling of such a prototype reduces design errors and integration risks.

The HDL under consideration is VHDL. Many engineering communities are using VHDL not only as a design documentation medium but also as a simulation medium [19, 20, 21]. VHDL supports a top-down approach by allowing high-level design abstractions as well as a bottom-up approach.

A single HDL such as VHDL captures the complete specification for design as well as all necessary constraints. Many other design files may be derived from a single

VHDL file as the design process progresses, but the designer only interacts with and changes the single VHDL file. A single HDL file provides the following advantages:

- Changing the control for a design involves changing only a single file rather than several files.
- Design file management is easier because the design specification is contained in one file and all other secondary design files are derived from the specification.
- Design consistency is easily maintained for design data within a single file.

Since most design files are derived from a single VHDL file, different modeling styles may produce different derived design files. Hence, the modeling style used is another important issue in the design process.

In this part, several important issues relating to modeling are presented. New FSM modeling techniques are proposed. These techniques are especially suitable for the modeling of large FSMs. By carrying out the actual design process starting from modeling to synthesis, it will be shown that different modeling techniques produce different synthesis results, thus highlighting the critical importance of modeling style. Benchmark results for 1-D convolution in a FPGA-based custom computing machine (CCM) and instrumentation system algorithm show trade-offs between modeling styles and their synthesized results.

Modeling techniques for systems and integration with VHDL are addressed in Part II, where system modeling experiences and the problems using VHDL are presented. In Chapter 2, modeling techniques for finite state machines (FSMs) are discussed. Parallel algorithms/architectures can be modeled using VHDL; this technique, along

with several examples, is presented in Chapter 3. In Chapter 4, the modeling experience in a CCM is considered. Finally, the relationship between the modeling styles and the synthesis results are given in Chapter 5. In Chapter 6, conclusions for Part II are presented.

Chapter 2

Finite State Machine

Almost all micro-electronic circuits are *sequential circuits*, i.e., they contain memory or storage elements in the form of flip-flops or latches as well as combinational circuitry. These sequential circuits are most often modeled using Finite State Machines (FSMs). A FSM is a mathematical model of a system with discrete inputs, discrete outputs, and a finite number of internal states. FSMs are very important in the micro-electronic devices, especially since the control circuitry of the system are usually implemented as FSMs. The choice of modeling technique is important because different models create different synthesis results.

In this chapter, a modeling technique for FSMs using VHDL is presented. The application of this technique to model systems has been demonstrated [22, 23, 24]. The actual machine selected to model FSM is the *Remote I/O Module* in the *SINCGARS Radio* [25, 26]. Figure 2.1 shows a block diagram of the *Remote I/O Module* in the *SINCGARS Radio*. Using VHDL, the current system, which uses a microprocessor, has been retargeted to a behavioral description of an FSM. The modeling technique

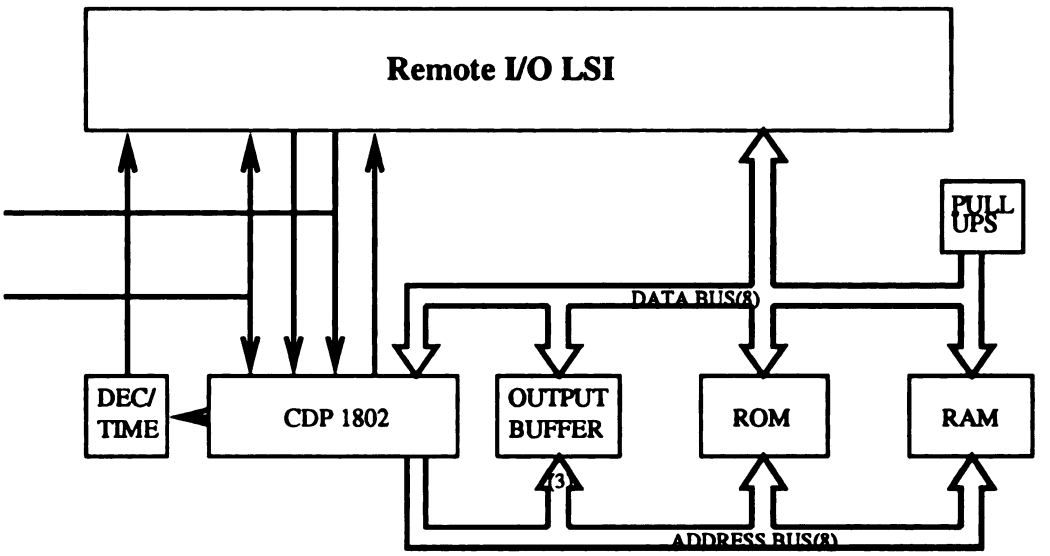


Figure 2.1: Block Diagram

has been successfully used to implement the SINCGARS circuit [19, 22].

The SINCGARS circuit was chosen for the following reasons. Since the final FSM is large, it provides an appropriate demonstration of the suitability of this approach to large FSMs. Furthermore, since the existing circuit has already been implemented and tested, testing the new FSM after replacement of the existing microprocessor is relatively less problematic.

2.1 Remote I/O Circuit Functionality

The current implementation of the *Remote I/O Module* consists of the *CDP 1802 microprocessor, ROM, RAM, OUTPUT BUFFER, DEC/TIM,* and the *REMOTE INTERFACE* as shown in Figure 2.1. The microprocessor is used in the *SINCGARS Receiver-Transmitter(RT)* to control its status. It controls the inputting, decoding, outputting, and encoding of remote control words sent from the other radio mod-

ules. Thus, the radio can be controlled from a distance via a physical wire. The *ROM* contains the stored program to perform the function of operating the *CDP 1802*. The *RAM* is used for temporary storage during operation of the *CDP 1802* system. The *OUTPUT BUFFER* can be loaded with parallel data which can be shifted out in a serial data stream. Output clocks and gates are also provided. The *DECODER/TIMER* is designed to decode control signals to the microprocessor and to provide timing for the system. The *REMOTE INTERFACE* contains two groups of latches. One group of latch signals is fed to it by the *CDP 1802*. These signals are used for output signals to the *RT*. The other group of latches is clocked and continually latches the latest state of its input lines. Figure 2.2 shows the state diagram of the Remote I/O module. During the microprocessor *SCAN* routine shown in Figure 2.2, the latched signals are read by the microprocessor via the data bus. The signals are analyzed, and appropriate action is taken. The microprocessor also provides direct memory access to the *RAM*.

The *Remote I/O Module* controls the power on and initializes operation of the *RT* in the remote configuration. It provides a self-test routine that tests the RAM, ROM, and remote 2-wire controls. It also provides remote front-panel controls for the *RT*: *MODE, CHANNEL, RF POWER, FUNCTION, KEYBOARD CONTROL*, and *COMSEC*. The Remote I/O Module sets and maintains the operating mode associated with the remote control units as listed above, and sends and receives the control signal for both control words and baseband information. Control signals are provided to modulate or demodulate an *FSK* carrier for control words.

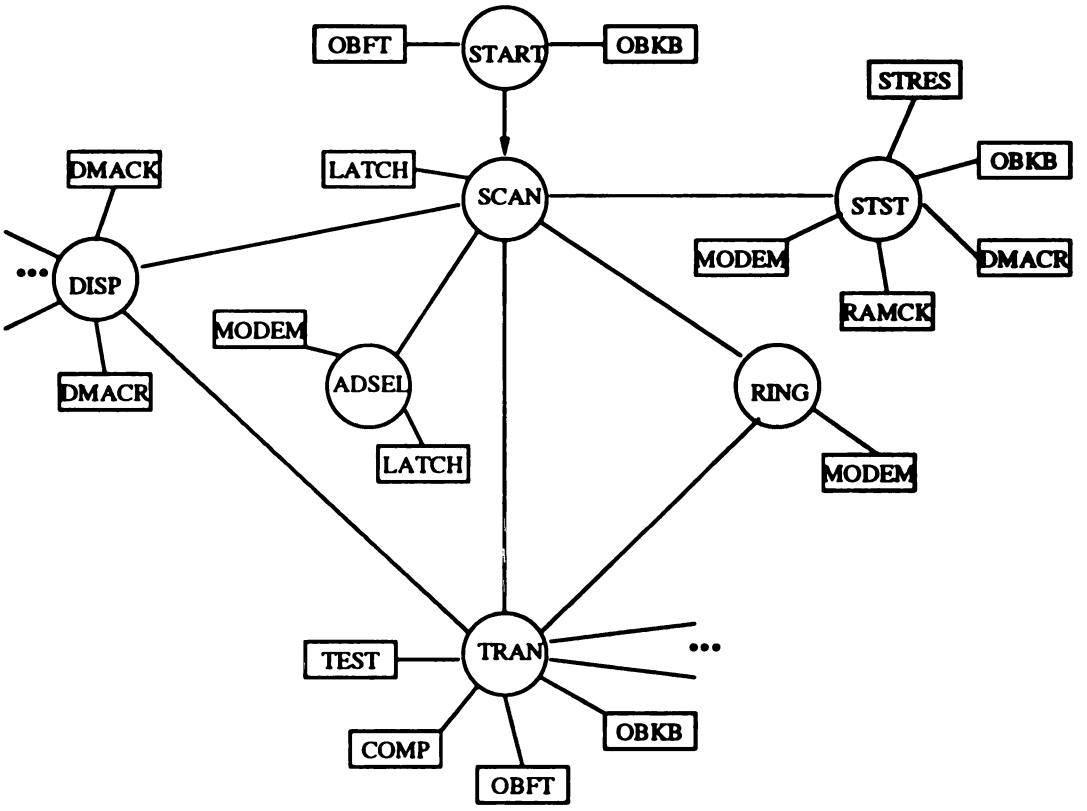


Figure 2.2: Remote I/O State Flow Diagram

```
state_assign:PROCESS
BEGIN
  -- see Figure:State Assign
END PROCESS;
fsm:PROCESS(present_state,x)
BEGIN
  CASE present_state IS
    WHEN 0 =>

      output <= '0';
      IF(x='0') THEN
        transition <= 0;
      ELSE
        transition <= 1;
      END IF;




    WHEN 1 =>
       ...
```

(A) Case Statement

```
PROCESS    -- state 0
  BEGIN
    WAIT ON clk;
    IF (present_state=0
         AND clk='0') THEN
      output <= '0';
      IF(x='0') THEN
        transition <= 0;
      ELSE
        transition <= 1;
      END IF;
    ELSE
      output <= NULL;
      transition <= NULL;
    END IF;
  END PROCESS;
PROCESS    -- state 1
   ...
```

(B) Process Statement

Figure 2.3: FSM Implementation

2.2 VHDL Modeling of the Finite State Machine

In VHDL modeling of FSMs, the **case** statement is widely used and cited in the literature [20, 27, 28]. An example is illustrated in Figure 2.3 (A) to model a simple FSM. However, when the FSM is large, it is useful to decompose the whole FSM into a number of small communicating FSMs so that each smaller FSM can then be modeled separately. These smaller FSMs are shown as circles in Figure 2.2. The modeling approach using **case** statements cannot handle this kind of decomposition effectively. In a proposed new approach which is described in this chapter, each small

FSM is represented as a single **process**, as shown in Figure 2.3 (B). There are several advantages to using the new approach: it is easy to decompose the large FSM into smaller FSMs; it is easy to manage smaller FSMs instead of handling one huge FSM; it is easy to update FSMs and maintain FSMs given the greater manageability of smaller FSMs.

The microcode in the *CDP 1802 processor* is modeled as a set of FSMs using VHDL. First, the overall behavioral description of the processor is represented with a VHDL behavioral description. This description is divided into a number of states in the finite state machine. In Figure 2.2, the circles represent main states, and the rectangles represent subroutines. These states and subroutines are further divided into states based on the generation of output signals, accessing the RAM, and waiting for the 640KHz clock input. The subroutines in the microprogram are also implemented as states.

The decomposed states in the original finite state machine are implemented using the **process** statements of VHDL. The FSMs are synchronized to the rising edge of the 640KHz clock. The interrupt routine is executed when the interrupt signal as well as the interrupt enable signal are both high.

The major disadvantage of using **process** is its extensive use of the **null** assignment. If the same signal is driven in more than one process block in VHDL, it must have a bus resolution function associated with this driver. In the new design, a signal (e.g., "transition," which selects the proper next state value) must be driven from only one process at a time, in other words, the rest of the drivers should be disconnected at the given simulation time. This resolution function is quite different

from those known as "Wired-Or," "Wired-And," and "Wired-X" functions, where all the drivers do not need to be disconnected. Their values are collected and used to calculate an output signal value of the function. In this design, if all the drivers except one are successfully disconnected, the resolution function simply returns the first driver's value because there is only one driver active at that time. In order to achieve this, a **null** assignment statement for each driver must be used.

Once the microprocessor is eliminated, the ROM is no longer necessary. The new block diagram is basically the same as the old one, except that there is no ROM, and CDP 1802 is replaced by FSM.

2.2.1 Basic Structure of State Modeling Using VHDL

There are several ways of representing the finite state machine in VHDL. The **case** statement is the most frequently used. An example of using a **case** statement is shown in Figure 2.3 (A). The other method, recommended here, uses a **process** statement. This method is illustrated in Figure 2.3 (B).

Each state has a fixed structure, as shown in Figure 2.4. One **process** represents one state. Each process waits for the changing value of *clk* (**wait on** statement). The next **if** statement controls the execution of the statements in that particular state. If the conditions are satisfied, the sequential statements inside of the **if** statement are executed. Each **process** can have any number of sequential statements. If the conditions are not satisfied, the **null** assignment statements are executed. At any given point of simulation time, only one process among all the processes actively

```
PROCESS
  -- local variables declarations here if needed
BEGIN
WAIT ON clk;
IF(present_state = ? AND clk = '0') THEN
   -- ? represents the current state number
   -- Sequential statements
  output <= new_output;      -- output assign
  transition <= next_state; -- state transition
   -- Sequential statements
ELSE
    output <= null;
    transition <= null;
END IF;
END PROCESS;
```

Figure 2.4: Basic State Implementation

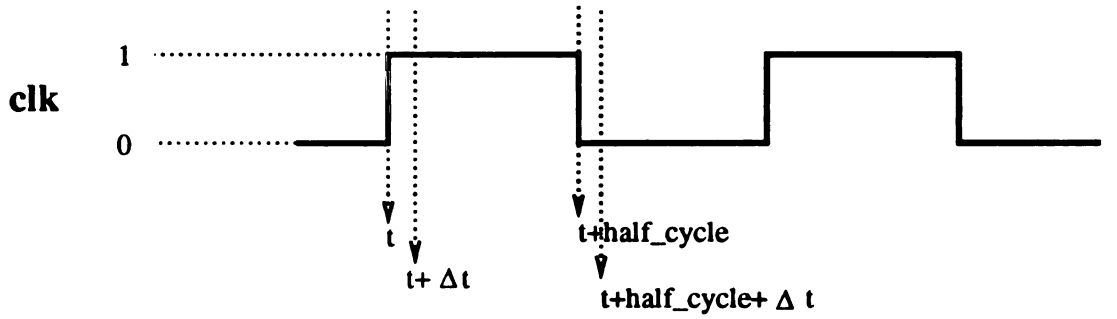


Figure 2.5: Timing Diagram

updates the new signal values; the rest of them simply disconnect all the signal drivers because of the *if* statement.

The new *present state* value is assigned at time **t**, which is the rising edge of the clock. For VHDL simulation, this value is available after Δt time interval. This timing diagram is shown in Figure 2.5. Thus, when the clock is low, the new value is already available to the process.

```
state_assign : PROCESS
BEGIN
  WAIT ON clk;
  IF(clk = '1' AND NOT clk'STABLE) THEN                    -- 1
   IF(interrupt_enable = '1' AND interrupt = '1') THEN     -- 2
    present_state <= 900; -- interrupt routine             -- 3
    state_save <= next_state;                              -- 4
   ELSE
    present_state <= next_state;                           -- 5
   END IF;
   -- Output assignments                                  -- 6
  END IF;
END PROCESS;
```

Figure 2.6: State Assign And Output Assign

2.2.2 State Transition and Output Generation

A state transition occurs at the rising edge of the clock, i.e., at the time **t** in Figure 2.5. This condition is checked by statement 1 of Figure 2.6. This new state value is available after Δt time delay, specifically, after the time $\mathbf{t+\Delta t}$ in Figure 2.5. Thus, when the clock is '0,' this new value is already available to all the processes in which the conditions can be checked. This is the case in the normal state assignment (statement 5) in Figure 2.6. However, if an interrupt request is made (conditions in statement 2), the next state value is saved (statement 4), and the flow of control goes to the state "900" (the beginning state of the interrupt handling routine) to serve the interrupt routine (statement 3).

Any output can be generated at the rising edge of clock at the same time that the state transition occurs. All the output signals can be assigned at statement 6.

2.2.3 Subroutine Handling

Subroutines are also implemented as FSM states. The execution of a subroutine is accomplished by assigning the beginning state number of the subroutine to *transition* and saving the return state number onto the stack. There is an independent process which handles the stack operation, shown in Figure 2.7. **RETURN** can be implemented by assigning the state "1000" to transition. At the state "1000," it simply *pops* the return address and makes the next transition available, as shown in Figure 2.7.

Since the `procedure` in VHDL cannot directly generate the output signal to the outside of the FSM, it is not possible to use procedures if there is output from the subroutines. This is one reason why subroutines are implemented by the FSM instead of using `procedures`. Each subroutine consists of several states, with the last state of each subroutine simply restoring the return state to the transition from the stack. Using the same technique as the main finite state machine, subroutine calling can be easily implemented.

The nested subroutine call uses the stack. Since the return address must be saved somewhere in order to return to the right place, the stack is an effective data structure. Every time a subroutine is called, the return address is saved onto the stack.

2.2.4 Interrupt Handling

If the interrupt request occurs while the program is executing in any of the states, as is shown in statement number 2 of Figure 2.6, the next state value is saved and the

```
stack_operation : PROCESS
  VARIABLE pointer : INTEGER := 0;
  VARIABLE stack : stack_ty := (others => 0);
BEGIN
  WAIT ON clk;
  IF( clk = '0') THEN
     IF(stack_operation = push) THEN
         stack(pointer) := return_state;
         pointer := pointer + 1;
         next_state <= transition;
     ELSIF(stack_operation = pop) THEN
         pointer := pointer - 1;
         next_state <= stack(pointer);
     ELSE
         next_state <= transition;
     END IF;
  ELSE
     next_state <=  null;
  END IF;
END PROCESS;

PROCESS  -- RETURN
BEGIN
  WAIT ON clk;
  IF(present_state = 1000 AND clk = '0') THEN
      stack_operation <= pop;
      memory_operation<= no;
  ELSE
      stack_operation <= null;
      memory_operation<= null;
  END IF;
END PROCESS;
```

Figure 2.7: Stack Operation and State 1000 RETURN

transition is changed to state "900," the start of the interrupt handling routine. At the end of the service, the saved state is restored to the signal ***transition*** in order to resume execution.

In the microprocessor, the execution of an instruction is divided into three phases: fetching, decoding, and executing the instruction. The interrupt request can occur during any of these three phases. When this happens, the current status is saved, the interrupt handling routine is invoked, and upon completion of the interrupt handling routine, the saved status is restored and the control is returned to the point where it was halted.

In VHDL simulation, no suitable way of implementing this situation can be found. Interruption can only occur when an instruction is finished executing. Although conditions are restricted so that interruption can occur after executing an instruction, this is not suitable for FSM modeling. Since several microinstructions were grouped into one state instead of allowing a single instruction per state, interruption should be allowed after finishing all instructions in a state.

The interrupt handler is implemented as is a finite state machine and can be viewed as a subroutine. When the interrupt request is given and when *interrupt_enable = '1,'* the next state transition is saved onto the signal variable called *state_save*, the new present state then becomes *900*, or the beginning of the interrupt handling routine. Upon completion of the interrupt handling routine, the next state is *1000*, which pops the saved state and restores the right transition information onto the signal variable *present_state*.

2.3 Discussion

How VHDL can be used to model the FSM and retargeting the current design into a new design have been shown. Since each state is modeled as one **process** in VHDL, this method has several advantages over use of a **case** statement. Using **process** statements gives better readability: each state may contain complicated sequential statements, decomposition can be easily performed, and only some states can be compiled and simulated independently within the whole system. Since this approach is modular, it is easy to manage smaller FSMs instead of handling one large FSM, and it is easy to update and maintain FSMs because only partial re-compilation is needed. This style, however, has the limitation of leading to excessive use of **NULL** assignment statements, which needs a bus-resolution function [22].

This FSM model is used to implement a hardware/software co-design system [24], which is explained in Chapter 5. Different synthesis results from several different FSM models are also summarized in Chapter 5.

Chapter 3

Modeling of Parallel Architectures and Algorithms

High-performance computing requires both parallel architectures and algorithms that work well on those architectures. To design a high-performance system, architectures and their suitable algorithms must also be designed. VHDL can be used to model parallel architectures and algorithms. For parallel algorithms, actual hardware with the desired architecture is usually not available and must be simulated. The tight coupling between architecture and algorithm requires a language that can describe both effectively.

For parallel architectures, two computing models are commonly used. In the single instruction, multiple data (SIMD) model, all of the processing elements (PEs) execute the same instruction synchronously. In multiple instruction, multiple data (MIMD) model, each PE has its own program and PEs only synchronize periodically. In either model, memory may be globally shared so that any PE can access any variable, or

memory may be local to each PE. In the latter case, PEs communicate by passing messages.

In general, a parallel algorithm on a specific architecture can be modeled as follows. At the top level, a structural description is used to create the processing elements and connect them using the chosen topology or architecture. A behavioral description in the architecture of each PE describes the activities of individual nodes. The memory of each node is modeled by variables within this **architectural** body of the VHDL description of the PEs.

For regular topologies, such as LINEAR ARRAY, RING, STAR, TREE, COMPLETELY CONNECTED, MESH, TORUS, and HYPER-CUBE [29, 30], VHDL **generate** statements work well. However, it may take some practice to correct the **port maps** using only the indices of the **generate** statements. Once a topology is developed, it usually can be easily extended to a larger system.

In the MIMD algorithm, individual PEs may be programmed independently. One way to handle this is to define different entities for each. This may preclude the use of the **generate** statement to describe the topology. Another solution is to lump the programs into one architecture and select a program using the **if** or **case** statement.

Figure 3.1 illustrates the general structure of a VHDL model for a SIMD parallel algorithm. The control unit for SIMD algorithms is modeled using a **process** at the top level. To be completely faithful to the SIMD model, most of the algorithm would have to reside in the control unit, and the PE architecture should only define the handling of individual instructions. However, individual instructions are of too fine a granularity. Instead, the algorithm must be divided into *meta-instructions*

```
...

-- Top Level

FOR i IN 0 TO ... GENERATE
    pe PORT MAP(local_clk, command,
                ...);
END GENERATE;

contrl: process
...
end contrl;

...

-- Individual Node
ENTITY pe IS
  PORT(local_clk : IN BIT;
       command : IN INTEGER;
       ...);
END pe;

ARCHITECTURE arch_pe OF pe IS
BEGIN
PROCESS(local_clk)
 VARIABLE k: INTEGER:=0;
 ...
 BEGIN
  CASE command IS
    WHEN ... =>
    ...
    WHEN ... =>
    ...
 END;
END;
```

Figure 3.1: General Structure of SIMD Algorithm Model

which handle sections of the code between necessary synchronization points. The **process** for the control unit issues these meta-instructions and the **process** for each PE executes the corresponding section of code.

3.1 Examples

In order to solve a given problem, finding a suitable algorithm and the best parallel architecture associated with the algorithm is needed. This is the driving program of each processing element. In MIMD, each PE executes different programs, while in SIMD, each PE executes the same instructions one at a time.

The first problem is finding the minimum among n different numbers, each of which is assigned to one node in a HYPERCUBE. The algorithm is a typical example of reduction. In each cycle k, every PE compares its data with its neighbor's data along the $k-th$ dimension, then saves the smaller number. After $log(n)$ cycles, every PE has the smallest number that is the solution. The VHDL code in Figure 3.2 implements the algorithm when $n = 8$.

The implementation of the algorithm in VHDL requires the modeling of a HYPERCUBE. A diagram of a HYPERCUBE with dimension 4 (called *4-Cube*) is shown in Figure 3.3, and the VHDL code for the *4-Cube* interconnection is shown in Figure 3.4. Here, *dim0* is used as the first dimensional connection such as X direction, *dim1* is used Y direction, *dim2* is used Z direction, and finally, *dim3* is used for the fourth dimension. The individual PE's id number is set by using *id(i)*, and *inidata(i)* is used to initialize the data for simulation if necessary.

```
LIBRARY WORK;
USE WORK.ALL;
ENTITY pe IS
  PORT(o0,o1,o2: OUT INTEGER;
       i0,i1,i2: IN INTEGER;
       id: IN INTEGER;
       local_clk: IN BIT;
       x: IN INTEGER);
END pe;

ARCHITECTURE arch_pe OF pe IS
BEGIN
PROCESS (local_clk)
 VARIABLE k: INTEGER:=0;
 VARIABLE data: INTEGER;
 TYPE int_array IS  ARRAY(0 TO 2) OF INTEGER;
 VARIABLE in_temp, out_temp: int_array;
 BEGIN
    in_temp(0) := i0;
    in_temp(1) := i1;
    in_temp(2) := i2;
    IF ((local_clk'EVENT) AND (local_clk = '0' )) THEN
         IF (k=0) THEN
                   data := x;
         END IF;
         out_temp(k) := data;
         k := k+1;
    END IF;
    IF ((local_clk'EVENT) AND (local_clk = '1' )) THEN
         IF ( in_temp (k-1) < data ) THEN
            data := in_temp(k-1);
         END IF;
    END IF;
    o0 <= out_temp(0);
    o1 <= out_temp(1);
    o2 <= out_temp(2);
END PROCESS;
END arch_pe;
```

Figure 3.2: Smallest Number Finding Problem

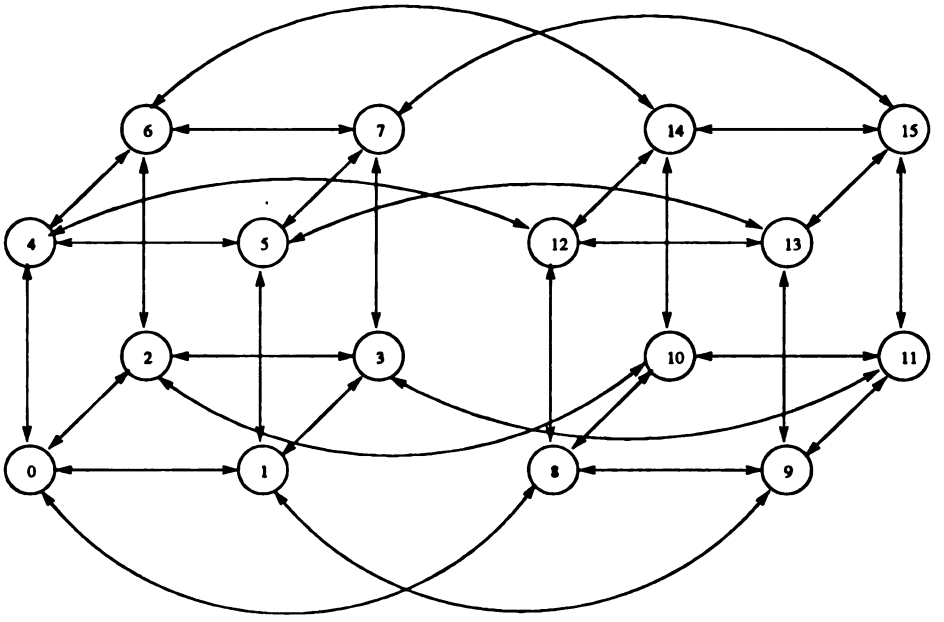


Figure 3.3: 4-Cube Connection

```
g: FOR i IN 0 TO 15 GENERATE
   id(i) <= i;
   p0: pec PORT MAP(dim0(i), dim1(i), dim2(i), dim3(i)  --(1) OUT's
                    dim0(i+(1-2*(i mod 2))),            --(2) IN's
                    dim1(i+(1-2*((i/2) mod 2))*2 ),
                    dim2(i+(1-2*((i/4) mod 2))*4 ),
                    dim3(i+(1-2*((i/8) mod 2))* 8 ),
                    id(i), clk, inidata(i));
END GENERATE;
```

Figure 3.4: 4-Cube Connection in VHDL

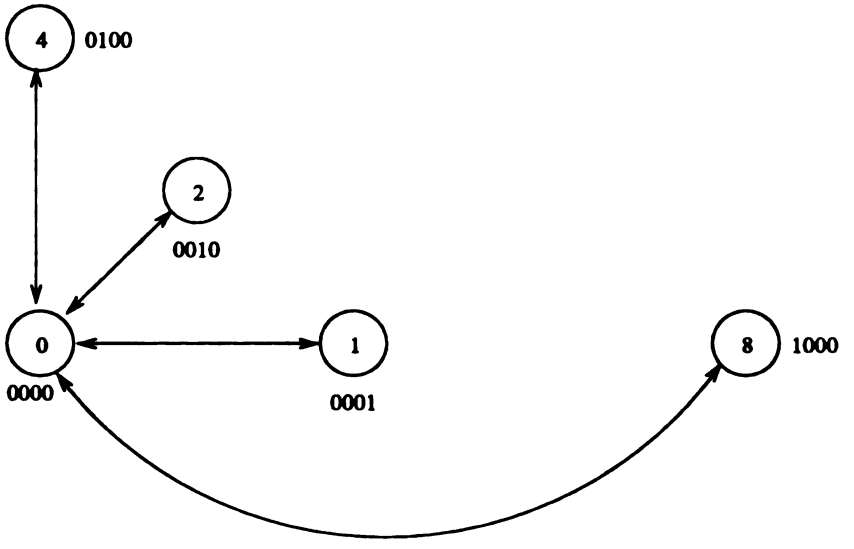


Figure 3.5: Bit Configuration of Connected PEs in 4-Cube

Since each PE has the same hardware configuration and executes the same program, there is only one PE description, which is saved in the design library. The network of PEs can be instantiated using **PORT MAP** statements. In addition, since the structure of the connection is regular, the **GENERATE** statement is used to instantiate all PEs. Each PE is connected to 4 PEs whose *id* differs only in one bit position. Figure 3.5 illustrates the connections of PE 0. The formula, `i+(1-2*((i/2**n) mod 2))*2**n`, in (2) of Figure 3.4, is used to find such a bit configuration. Here `i` represents the relative *id* number of each PE, and `n` represents the *n-th* bit position of *id*.

The next example implements *Batcher's Sorting Algorithm* on a TORUS network. The VHDL code segment shown in Figure 3.6 is an implementation of the algorithm based on Nassimi's paper [31]. Each PE receives commands from the top module, which may be regarded as an SIMD controller, and responds to those commands. The commands are very simple, such as *shift_data_left, _right, _up*, and *_down*, or

change_data between different components within the same PE. Operations related to a single PE are implemented by sending corresponding commands and parameters to each PE.

A *5-by-5* TORUS is shown in Figure 3.7, and its VHDL code is shown in Figure 3.8. Again, the regularity of TORUS permits the use of a **generate** statement to instantiate all the components. In the VHDL code, the signals `con0`, `con1`, `con2`, and `con3` are used to connect each PE in all four directions. The wrap-around is implemented using integer **mod** functions.

The second program shown in Figure 3.6 describes the implementation of an actual SIMD PE. A PE receives a command, such as *left_shift* or *change0and2*. The execution of this instruction is synchronized at the control signal *local_clk*.

Note that the VHDL implementation of the second problem more closely model the behavior of SIMD machines than the implementation of the first problem. In SIMD machines, a PE has no program to run. Each PE only responds to instructions sent from the controller or control signals such as clock. However, this implementation in the second example still uses some *meta* command characteristics. For example, the command *change0and2* contains more than one instruction. In the first example (Figure 3.2), however, each PE contains most of the algorithmic code. The controller sends only synchronization signals as opposed to instructions.

Describing MIMD machines is not an easy task, especially when the machine is dynamically configured. Each PE may have a different program (instruction) to run; thus, it may not be feasible to use the **generate** statement. Each PE must be coded independently and connected separately.

```
-- ...  deleted
ARCHITECTURE arch_pe OF pe IS
BEGIN
PROCESS(local_clk)
-- ...  deleted
  rowin_right_temp := rowin_right;
  CASE command IS
    WHEN left_shift =>
      IF(local_clk = '0' AND local_clk'EVENT) THEN
        IF(colid MOD (2*par1) + par2 <= 2*par1) THEN
          rowout_left_temp := pe1;
        END IF;
      END IF;
      IF(local_clk = '1' AND local_clk'EVENT) THEN
        IF(par2 + colid MOD (2*par1) - par1 >= 0) THEN
          pe1 := rowin_right_temp;
        END IF;
      END IF;
    -- ...  deleted
    WHEN change0and2 =>
      IF(local_clk = '0' AND local_clk'EVENT) THEN
        IF(rowid MOD (2*par1) < par1) THEN
          IF(sort_order = 1) THEN
            IF(pe2 < pe0) THEN
              temp := pe0; pe0 := pe2; pe2 := temp;
            END IF;
          END IF;
          IF(sort_order = -1) THEN
            IF(pe2 > pe0) THEN
              temp := pe0; pe0 := pe2; pe2 := temp;
            END IF;
          END IF;
        END IF;
      END IF;
    -- ...  deleted
    WHEN OTHERS => NULL;
  END CASE;
  -- ... deleted
  result <= pe0;
  rowout_left := rowout_left_temp;
END PROCESS;
END arch_pe;
```

Figure 3.6: PE Behavior of Batcher's Sorting Algorithm

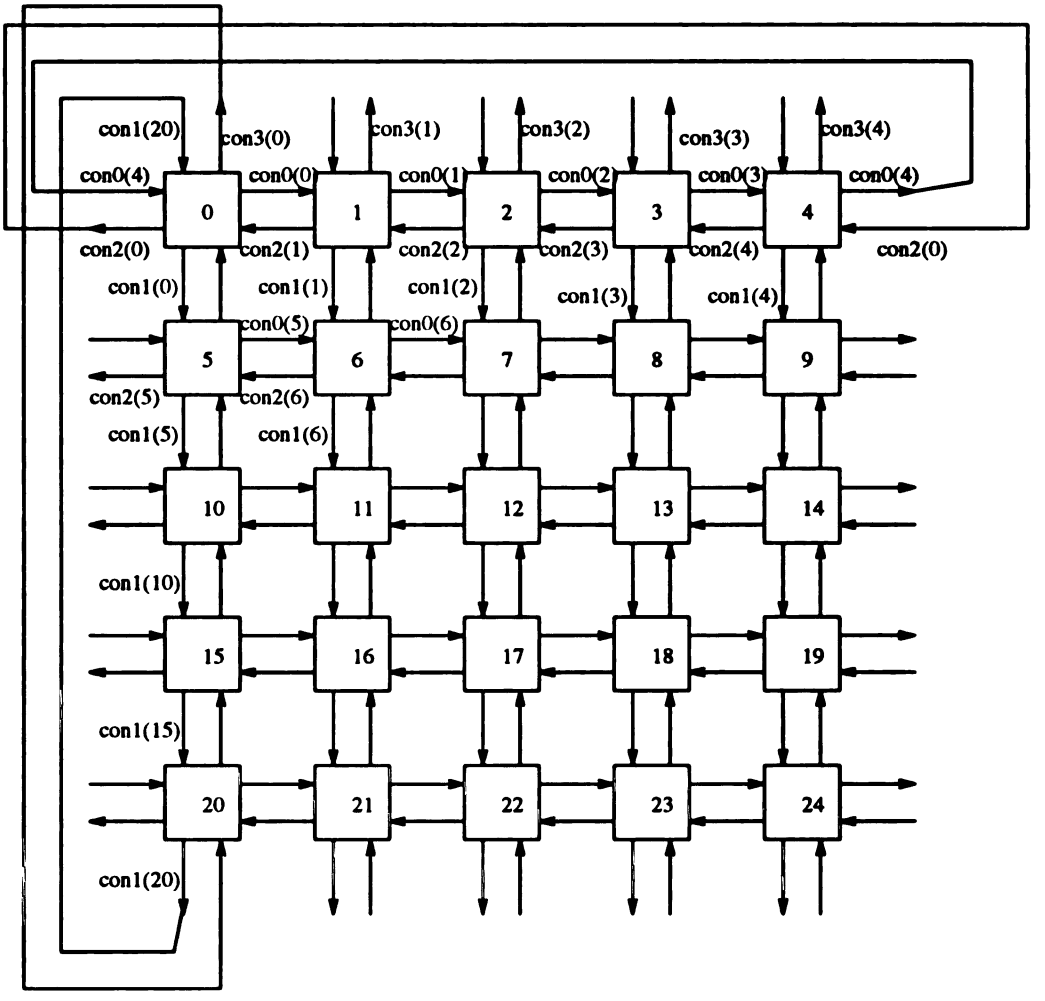


Figure 3.7: 5X5 Torus Connection

```
g:FOR i IN 0 TO 4 GENERATE
  g1: FOR j IN 0 TO 4 GENERATE
      id(i*5+j) <= i*5+j;
      p0: pec PORT MAP(con0(i*5+j),          --(1) EAST  out
           con1(i*5+j),                      --(2) SOUTH out
           con2(i*5+j),                      --(3) WEST  out
           con3(i*5+j),                      --(4) NORTH out
           con0(((i*5+j-1) mod 5) + i*5),    --(5) WEST  in
           con1(((i-1)*5+j) mod 25),         --(6) NORTH in
           con2(((i*5+j+1) mod 5) + i*5),    --(7) EAST  in
           con3(((i+1)*5+j) mod 25),         --(8) SOUTH in
           id(i*5+j), clk);             --(9) PE id and Clock
   END GENERATE;
END GENERATE;
```

Figure 3.8: TORUS Connection

VHDL has been shown to be powerful enough to describe various parallel architectures and their timing behaviors. Using VHDL, parallel algorithms can be simulated and the performance of parallel architectures measured. To model a specific parallel application and synthesize to a specific parallel hardware, Splash 2 was used as a signal processing application. This application is covered in the next chapter.

Chapter 4

Signal Processing in a Custom Computing Machine

Signal processing applications are computation-intensive, primarily because of the large amount of data to be handled in a very short time. Single processor machines cannot attain the desired performance, parallel machines and application-specific integrated circuits (ASICs), therefore, are used to obtain the desired speed. However, these options are costly, moreover, once the ASIC is built, the design is very difficult to be changed.

Field Programmable Gate Arrays (FPGAs) have gained considerable attention recently because of their reconfigurability. FPGAs allow a new form of computing where the architecture of a computer may evolve over time, changing to fit the needs of each application it executes. With an FPGA-based machine, architecture can be tailored to meet the desired performance for a given application.

The reconfigurable architecture, which has been adopted, is Splash 2, developed

and built by the Supercomputing Research Center (SRC). Splash 2 is a special purpose attached parallel processor having processing elements (PEs) based on user programmable Xilinx 4010 FPGA chips. The Splash 2 system consists of a Sun SPARCstation as a host, an interface board, and Splash array boards ranging from one to sixteen boards. Each array board consists of 16 PEs with linear connections as well as a reconfigurable crossbar interconnection between PEs.

The Splash 2 system can be used to enhance existing applications by using its multiple PEs. The applications should be modeled appropriately to the Splash 2 system. The Splash 2 system does not have any fixed instructions; any sets of instructions can be designed in this system. Thus, modeling is very important step in the design process when using the Splash 2 system.

VHDL is used to model computational algorithms as well as architectures. Simulation is used to verify the parallel model. Then, from the VHDL description, the compiler retargets the algorithms and architectures into FPGAs. The role of VHDL in this system is multifaceted: it is used as both the specification (modeling) language and the implementation language. VHDL describes both algorithms within user applications and the necessary hardware to realize them (including processing units, memories, and interconnections) [32].

In this chapter, research work to model and synthesize a signal processing application using VHDL targeted to a parallel custom computing machine, Splash 2, is described. Different modeling approaches are greatly affected in the final synthesized results: this aspect is summarized in Section 4.4.

4.1 Splash 2

Splash 2 [33] is an attached special purpose parallel processor where each processing element is a user programmable FPGA chip. The architecture of Splash 2 can easily support parallel applications, such as systolic or data-parallel computations. Splash 2 has been developed and modified from the Splash 1 system [34], which consisted of a fixed size linear array of Xilinx 3090 FPGA chips. Splash 2 incorporates several improvements over Splash 1. Splash 2 is based on newer hardware technology, the Xilinx XC4010 FPGA. A crossbar has been added to connect PEs on a board in any pattern. The linear path was the only configuration in Splash 1. The programming environment of Splash 2 centers on VHDL, in place of SRC's Logic Description Generator (LDG) [35, 36, 37].

4.1.1 Splash 2 Architecture

Splash 2 is attached to a Sun SPARCstation host. Figure 4.1 shows the Splash 2 architecture [38, 39]. The host is connected to Splash 2 via an interface board. The host can read from and write to memories on the Splash processing boards via this interface board.

Each Splash 2 processing board has 16 processing elements, X_1 through X_{16}, with one special PE, X_0, controlling the data flow into the processor board. Each X_i is a PE built around a Xilinx 4010 FPGA chip. A crossbar connection can be programmed by X_0. The processing element organization is shown in Figure 4.2. Each PE has 512 KB of attached memory, which has a 16-bit word size. The host can access this

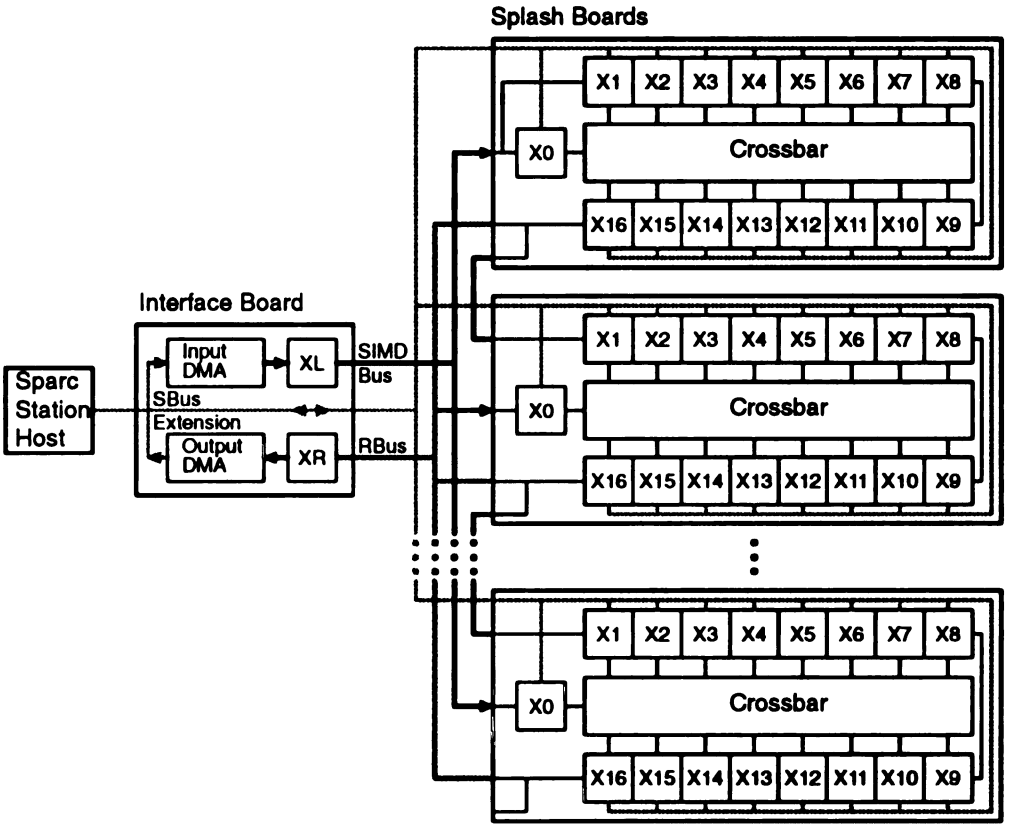


Figure 4.1: Splash 2 Architecture

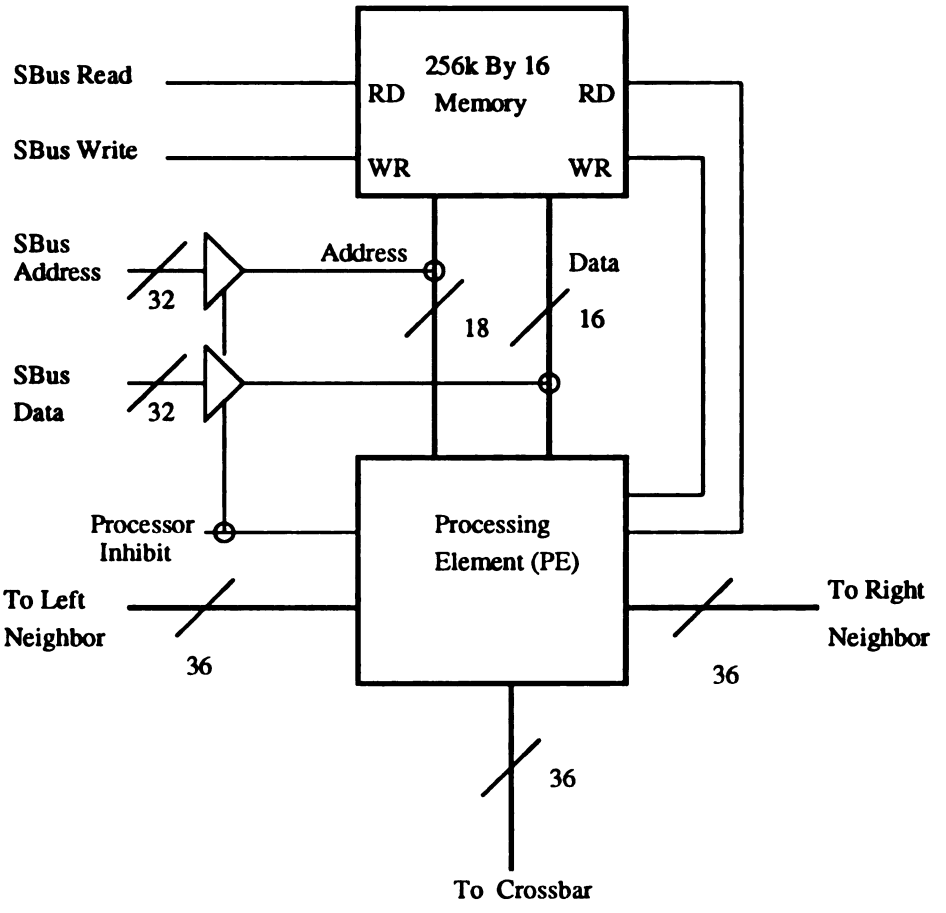


Figure 4.2: Processing Element (PE) in Splash 2

memory via the SBus.

Each PE can communicate using the SIMD Bus (left-right neighbor data paths) or the crossbar. Data can be broadcast using the crossbar. The Splash 2 architecture supports systolic and pipeline modes of computation, SIMD, or data-parallel mode (PEs execute the same instructions on different data streams), or even MIMD mode (PEs execute different instructions on different data streams). The Splash 2 system can run at a maximum clock speed of 40 MHz where the maximum is limited by the FPGA technology. The actual operating speed is determined when the FPGA logic is synthesized.

4.1.2 Programming Splash 2

The programming environment for Splash 2 is based on VHDL [14, 40]. The behavioral description is analyzed, simulated, and synthesized onto Xilinx FPGAs. An application for Splash 2 is developed by writing its behavioral description in VHDL, and the description is iteratively refined and debugged with the Splash 2 system simulator. After the description is verified to be functionally correct by simulation, it is translated into a Xilinx net list form. The net list is then mapped onto the FPGA architecture by an automatic partition, placement, and routing tool to form a loadable FPGA object module. A static timing analysis tool is then applied to the object module to determine the maximum operating speed.

To program Splash 2, each of the PEs should be programmed, i.e., X_0 through X_{16}, and the host interface. The host interface is responsible for data transfers between host and the Splash 2 board. For this purpose, the system provides a C-language interface and C programs are written for the host's tasks.

4.2 Convolution

An important class of signal and image processing algorithms is based on the convolution of two signals. For analog one-dimensional signals $f(t)$ and $g(t)$, the convolution $h(t)$ is defined as

$$h(t) = \int_{-\infty}^{+\infty} g(x)f(x-t)dx \tag{4.1}$$

1. **Input:** A 1-dimensional vector f(t), a mask vector g(t).
2. **Output:** A 1-dimensional result vector h(t).
3. **Begin**
   Assume k PEs are available.
   The i^{th} PE holds the g(i) mask value.
   On each PE:
   Receive from left neighbor: signal value f(i), partial sum S(i-1).
   Compute new partial sum S(i) = S(i-1) + f(i) * g(i).
   Send signal value f(i) and partial sum S(i) to right neighbor.

End.

Figure 4.3: Sequential Algorithm of Convolution

Discrete-time one-dimensional signals reduce to the following equation:

$$h(t) = \sum_{-\infty}^{+\infty} g(x)f(x-t) \tag{4.2}$$

Further, if the signal $g(t)$ is a finite-time duration signal (called the mask signal), then the summation range changes, so that

$$h(t) = \sum_{x=1}^{+k} g(x)f(x-t) \tag{4.3}$$

where k is the mask size.

On a sequential machine, convolution is easily implemented, as shown in Figure 4.3. However, a sequential computer may not be practical when convolution needs to be done in real-time for a large number of data points. For parallelizing a convolution computation, two approaches can be taken [41]: (i) data parallel computing and (ii) systolic computing. Data parallel computing uses a divide-and-conquer approach to deal with the large amount of data. Usually $f(t)$ has a large number

of data points (spread over a large time domain) compared to the mask signal $g(t)$. Hence, a set of processors can be used to compute on shorter segments of the data in parallel. This computational model assumes that each PE is powerful enough to carry out all computations and that signal values are already available at each PE. If this latter assumption is not the case, then the data path from host to PEs becomes a bottleneck, preventing the distribution of data to the PEs. This problem can be overcome by using a systolic approach. This requires only that a single data path exist between the host and k PEs and that the PEs be powerful enough to perform the add and multiply operations.

A set of PEs are used as a linear array. The basic convolution algorithm translates into the above systolic algorithm. The input is fed into the PE array at the left input of the first PE with the partial sum initialized to zero. At the output of the last PE, the final result is obtained.

Clearly, the algorithm outputs one result every clock cycle after the initial pipeline latency. The overall model is schematically described in Figure 4.4. If the number of PEs available is smaller than the number of mask values, several virtual PEs to a physical PE should be mapped, in which case, the PEs need wider communication paths and require more cycles to produce results.

4.3 VHDL Implementation

In this section, the implementation of a 1-D convolution using VHDL is described. In the implementation of this convolution, the PEs are configured as a linear array and

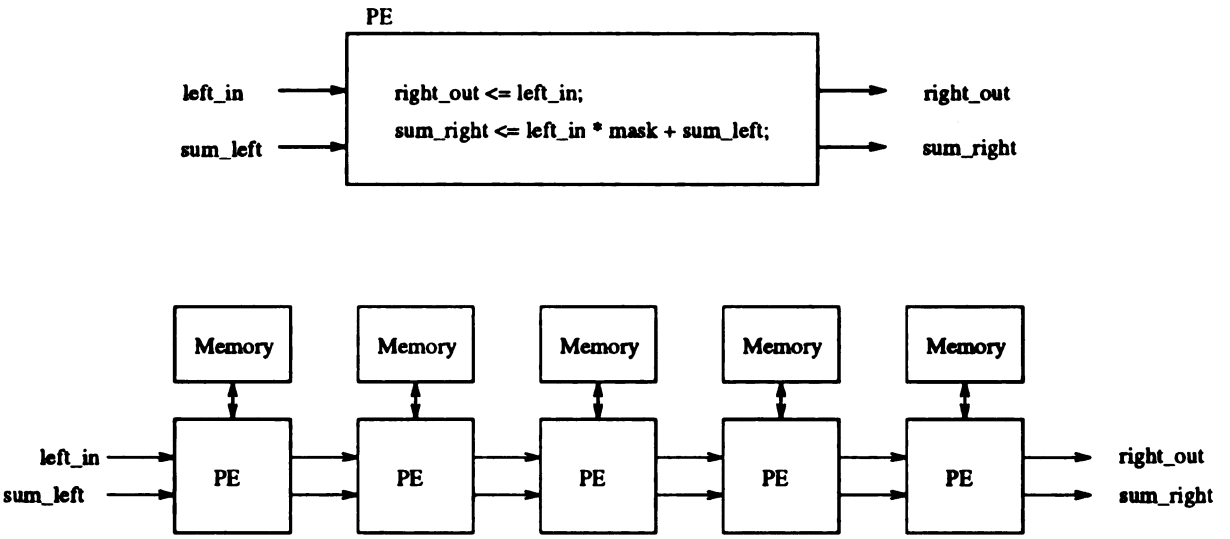


Figure 4.4: PE Behavior and Configuration for 1-D Convolution ($k = 5$)

a systolic method is used. In a 1-D convolution, each signal value (ranging from 0 through 255 in the applications being targeted) is input into the left-most PE at each clock cycle, and this input is multiplied by the mask value. For each signal value, the input value and partial sum are passed to the next PE to the right for accumulation.

4.3.1 1-D Convolution

A signal value, *left_in*, is received at each clock cycle and is multiplied by the mask value, *mask*. The partial sum, *psum*, is simultaneously received and is added to this newly-formed product. The signal value and the new partial sum, *sum*, are then passed to the right PE for accumulation. The VHDL program representing this PE behavior is shown in Figure 4.6. The configuration is shown in Figure 4.5.

In the program, the calculation of the accumulation is shown at line 4 in Figure 4.6. Synchronization with the clock is achieved by waiting for the clock transition (line 1) in Figure 4.6.

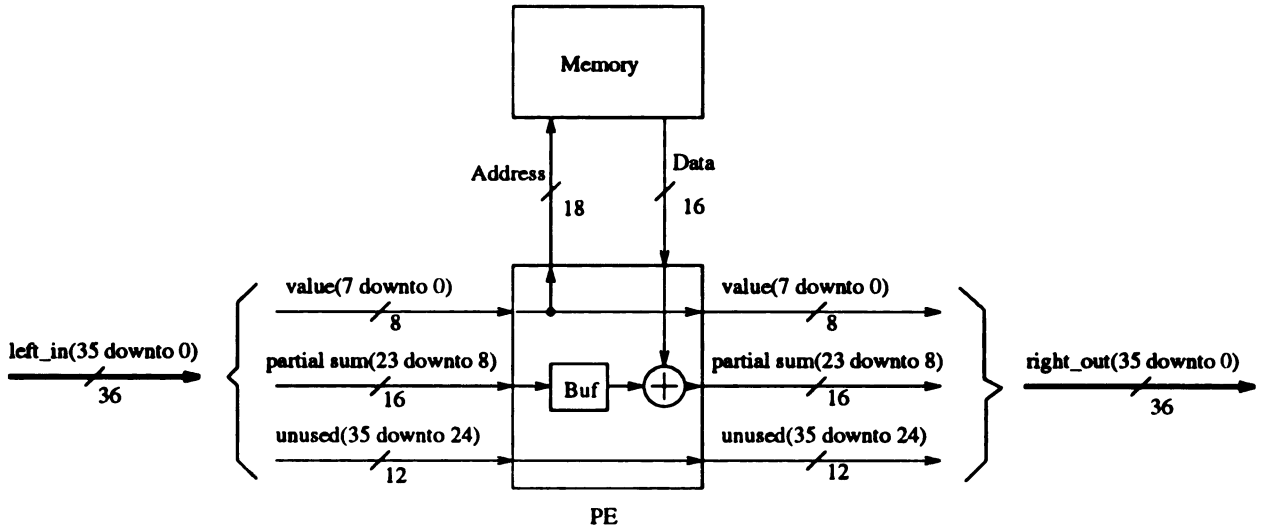


Figure 4.5: Splash PE Input/Output Data Paths

```
WAIT ON valid_clk;                          -- 1
psum := sum_left;                           -- 2
IF (left_in>0) AND (psum>=0) THEN           -- 3
   sum := mask * left_in + psum;            -- 4
ELSE
   sum := psum2;
END IF;
right_out <= left_in;
sum_right <= sum;
```

Figure 4.6: Behavioral Modeling

```
ENTITY Xilinx_Processing_Element IS
  GENERIC(
    bd_id : INTEGER := 0;           --  Splash Board ID
    pe_id : INTEGER := 0);          --  Processing Element ID
  PORT (
    XP_Left : INOUT DataPath;       --  Left Data Bus
    XP_Right: INOUT DataPath;       --  Right Data Bus
    XP_Xbar : INOUT DataPath;       --  Crossbar Data Bus
    XP_Xbar_EN_L : OUT BIT_VECTOR(4 DOWNTO 0);
                                    --  Crossbar Enable (low-true)
    XP_Clk  : IN    BIT;            --  Splash System Clock
    XP_Int  : OUT   BIT;            --  Interrupt Signal
    XP_Mem_A: INOUT MemAddr;        --  Splash Memory Address Bus
    XP_Mem_D: INOUT MemData;           --  Splash Memory Data Bus
    XP_Mem_RD_L         : INOUT RBit3; --  Memory Read Signal
    XP_Mem_WR_L   : INOUT RBit3;       --  Memory Write Signal
    XP_Mem_Disable: IN    BIT;         --  Splash Memory Disable
    XP_Broadcast  : IN    BIT;         --  Broadcast Signal
    XP_Reset      : IN    BIT;         --  Reset Signal
    XP_HS0        : INOUT RBit3;       --  Handshake Signal Zero
    XP_HS1        : IN    BIT;         --  Handshake Signal One
    XP_GOR_Result : INOUT RBit3;       --  Global OR Result Signal
    XP_GOR_Valid  : INOUT RBit3;       --  Global OR Valid Signal
    XP_LED        : OUT   BIT);        --  LED Signal
END Xilinx_Processing_Element;
```

Figure 4.7: PE Entity Description

4.3.2 Splash 2 Implementation

The 1-D convolution program is ported to the Splash simulator in order to simulate and synthesize for the Splash architecture and for Xilinx FPGAs. The entity description [36] of each PE is shown in Figure 4.7. The signals are self-explanatory and are shown schematically in Figure 4.2.

The behavioral code for a 1-D convolution using VHDL cannot be directly synthesized into a Xilinx FPGA chip, since an FPGA consists of a finite, and therefore,

```
WAIT until XP_Clk'EVENT AND XP_Clk = '1';                -- 1
-- ...
psum := psum1;
psum1 := bvtoi(left_in(23 downto 8));                    -- 2
--    sum := mask*bvtoi(left_in(7 downto 0))+psum;       -- 3
Address(7 downto 0) <= left_in(7 downto 0);              -- 4
sum := bvtoi(Data)+psum;                                 -- 5
right_out(7 downto 0) <= left_in(7 downto 0);            -- 6
right_out(23 downto 8) <= itobv(sum,16);                 -- 7
```

Figure 4.8: PE Behavioral Description

limited number of logic devices. The logic synthesized directly from the VHDL description includes 8-bit multiplication, which consumes too many gates to fit into the Xilinx chip. To reduce the logic requirements, the multiplication operation is converted into a table look-up operation using the memory of the PEs. The VHDL code segment for the Splash PE is shown in Figure 4.8. A diagram of the PE input/output data paths is shown in Figure 4.5.

Each PE is synchronized by the rising edge of the clock, **XP_Clk** (line 1) of Figure 4.8. The effect of lines 4 and 5 is shown in the comment of line 3: a memory look-up is used in place of multiplication. Since each signal is 8 bits wide, there are 256 distinct values possible for each data point. The 8-bit input is shown in line 4. It is used as an address of a 256-word look-up table, which holds the results of the 256 values times the mask value (a single constant for that PE). The 256 multiplications can be calculated in advance and load these values into the PE's memory. The loading of these values is done from the host of the Splash system before the convolution execution starts. The signal value is used as the look-up table address.

Once the *Address* is set to a certain value (line 4), the value in the memory location pointed to by *Address* is loaded into a variable *Data* at the next clock cycle. Thus the multiplication result is available at the next clock cycle in *Data* (line 5). One buffer is used for restoring a temporary result (*psum1* at line 2) since *Data* is available one clock cycle later. Before the partial sum is calculated, conversion of the signal of the bit-vector type into an integer type is needed. This is done by using the function *bvtoi*. Similarly, *itobv* is used to convert an integer to a bit-vector. The input signal value and partial sum are packed and passed to the right neighbor (lines 6 and 7).

Once the PE program is complete, the Splash system can be configured. This is done by programming the VHDL top model. A portion of this code is shown in Figure 4.9. A predefined interface board configuration is used. By assigning the generic constants such as *input.dat*, the system can be tailored (line 1). Line 2 specifies one Splash board. A specific interconnection between PEs is obtained by loading the initial configuration from a file, here named *xcrossbar* (line 3). Each PE is configured by using a predesigned component. Line 4 marks the section of code for PE 1, line 5 for PE 2; the rest of the PEs are defined similarly.

4.4 Experimental Results

Splash simulation indicates the correct behavior of the model, as determined by analyzing Figure 4.10, the simulation waveform. The synthesized result achieves a clock rate of 18.5 MHz for the 1-D convolution, as shown in Figure 4.11.

The timing results show that different operations can run at various maximum

```
CONFIGURATION top OF Splash_System IS
-- ...
  USE ENTITY Interface.Interface_Board(Structure)    -- 1
    GENERIC MAP (Input_file1  => "input.dat",
                 Output_file1 => "result.dat",
                 File_Type => Hex,  Clock_Freq => 20);
-- ...
  USE ENTITY S2Board.Splash2_Boards(Structure)       -- 2
    GENERIC MAP (Number_Of_Boards => 1);
-- ...
  USE ENTITY S2Board.Splash_Crossbar(Behavior)       -- 3
    GENERIC MAP (Config_File => "xcrossbar");
-- ...
FOR xparts(1)     -- PE 1                             -- 4
  FOR ALL : Xilinx_Processing_part
     USE ENTITY WORK.Xilinx_Processing_Part(conv_1d);
  END FOR;
  FOR ALL : Memory_Part
     USE ENTITY S2Board.Memory_Part(Dynamic)
       GENERIC MAP (Load_File => "lookup01.dat");
  END FOR;
END FOR;
FOR xparts(2)     -- PE 2                              -- 5
  FOR ALL : Xilinx_Processing_part
     USE ENTITY WORK.Xilinx_Processing_Part(conv_1d);
  END FOR;
  FOR ALL : Memory_Part
     USE ENTITY S2Board.Memory_Part(Dynamic)
       GENERIC MAP (Load_File  => "lookup02.dat");
  END FOR;
END FOR;
-- ...
```

Figure 4.9: Top Module Configuration Description

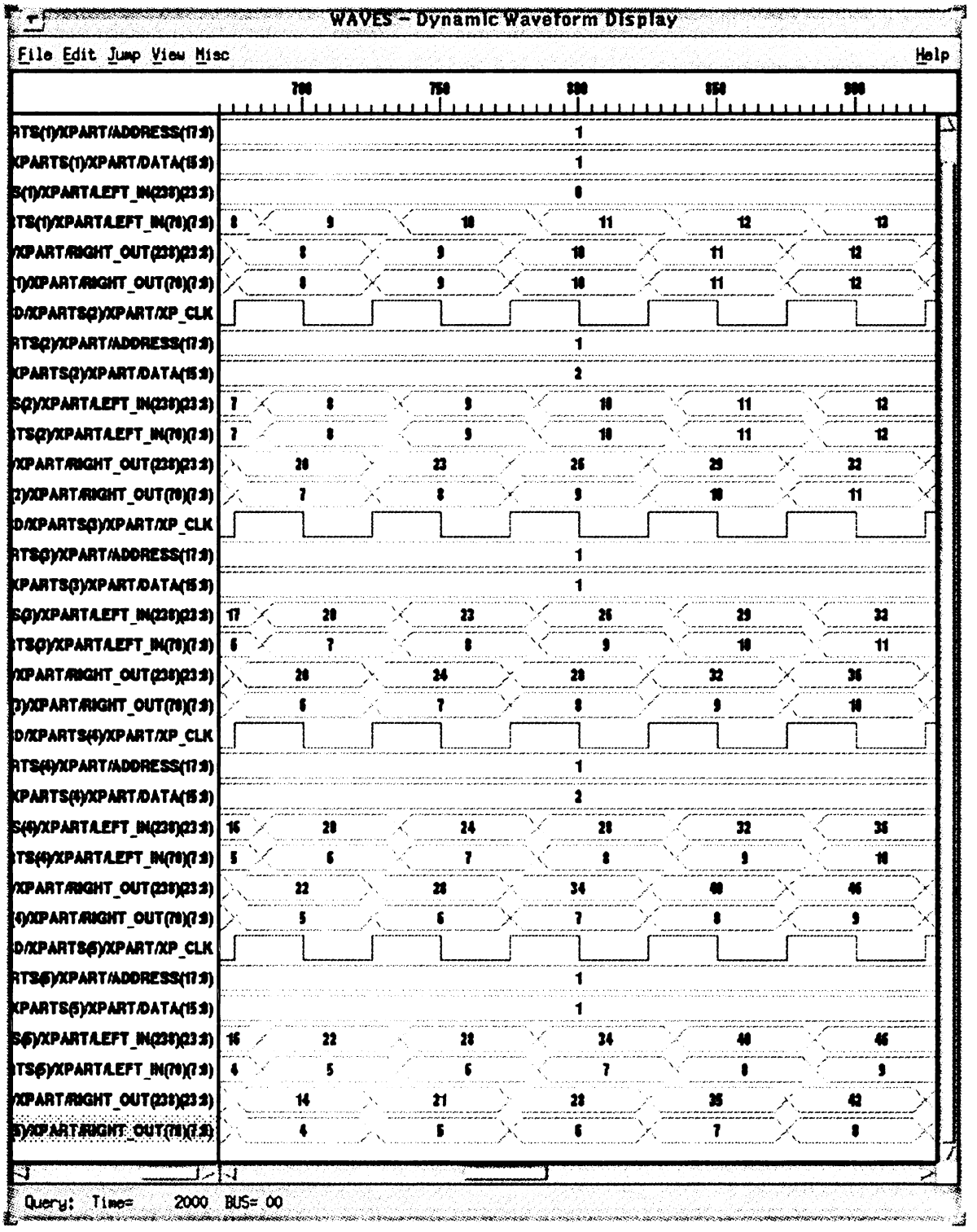


Figure 4.10: Simulation Result for 1-D Convolution

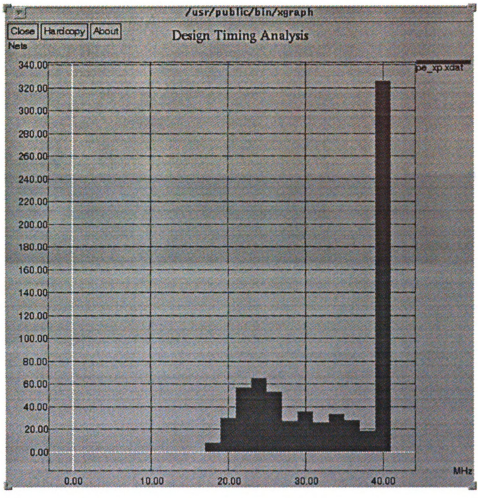


Figure 4.11: Timing Result Using Memory Lookup

clock rates. For example, in Figure 4.11 several operations can run at a maximum clock rate of 18.5 MHz, while other operations can run at a maximum rate of 40 MHz. The 18.5 MHz clock rate becomes the maximum operating speed. Figure 4.12 shows a different timing profile obtained by using a different PE implementation. Specifically, multiplication is performed as a series of additions. For non-negative mask values, the signal value is repeatedly added a number of times equal to the mask value. Using this method, the performance decreases to a maximum clock rate of 9.5 MHz, as shown in Figure 4.12.

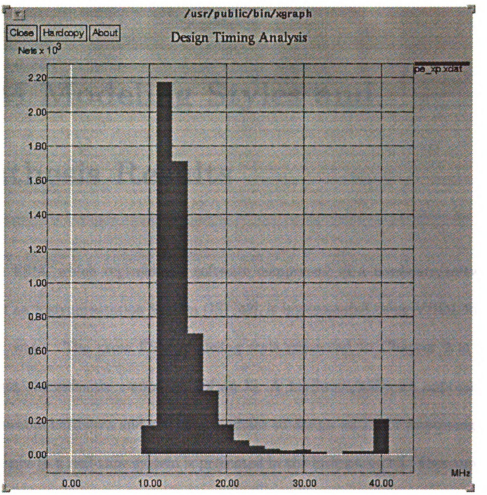


Figure 4.12: Timing Result Using Loop Statement

Chapter 5

FSM Modeling Styles and Synthesis Results

A single FSM, which represents a software component of a hardware/software co-design of an Instrumentation System (IS) [42], is implemented using VHDL in several different ways. The same FSM modeling style presented in Chapter 2 is used to implement the software component of an IS. A hardware/software approach using reconfigurable hardware and software modules to design an IS for improving system performance in a real-time system is presented in the literature [24]. This approach is different from well-known hardware and hybrid (hardware and software) monitoring approaches. The Splash 2 custom computing machine (CCM) is used, transferring the critical parts of the analysis modules of an IS to programmable logic to ensure that other real-time analysis and optimization tasks are not affected by the latency of these modules.

Three different modeling styles were implemented and to synthesize each design.

PE #	Max. Clock Rate in MHz (Clock Period)	# of Clock Cycles Needed for each Iteration	# of States Needed	% Utilization of Available CLBs
0	18.9 (52.8 nsec)	13	21	73
1-16	17.9 (56 nsec)	13	28	50

Table 5.1: Synthesis Statistics Critical to the Performance and Realization of the Hardware Subsystem

In this chapter, the different synthesis results obtained from the different modeling styles are presented. State diagrams are used to model the behavior of the subsystems in VHDL. Figure 5.1 shows the state diagram for each PE in Splash 2. Commercial synthesis tools from Synopsis and Xilinx, integrated into the Splash 2 environment directly, use this representation to generate logic configurations for the PEs in Splash 2. This synthesis process provides various statistics regarding the design of the hardware subsystem critical to its performance, for instance, timing information. Different VHDL representations of the same function may result in different timing behaviors.

The maximum clock rate at which each PE can operate is represented in the second column of Table 5.1. PE 0 can operate slightly faster than the rest of the PEs; however, the overall performance is restricted by the slower PEs, i.e., PEs 1 to 16. Thus PEs 1 to 16 affect the final operating speed, 17.9 MHz (56.0ns), though PE 0 can operate at a maximum speed of 18.9 MHz. The number of cycles needed (13 cycles) for handling each event record is the same for all PEs (as shown in the column three of the table), despite the total number of states in PE 0 and PEs 1 to 16 are 21 and 28, respectively, shown in the column four. The last column shows the utilization rate of the configurable logic blocks (CLBs) of Xilinx 4010 chips. PE 0

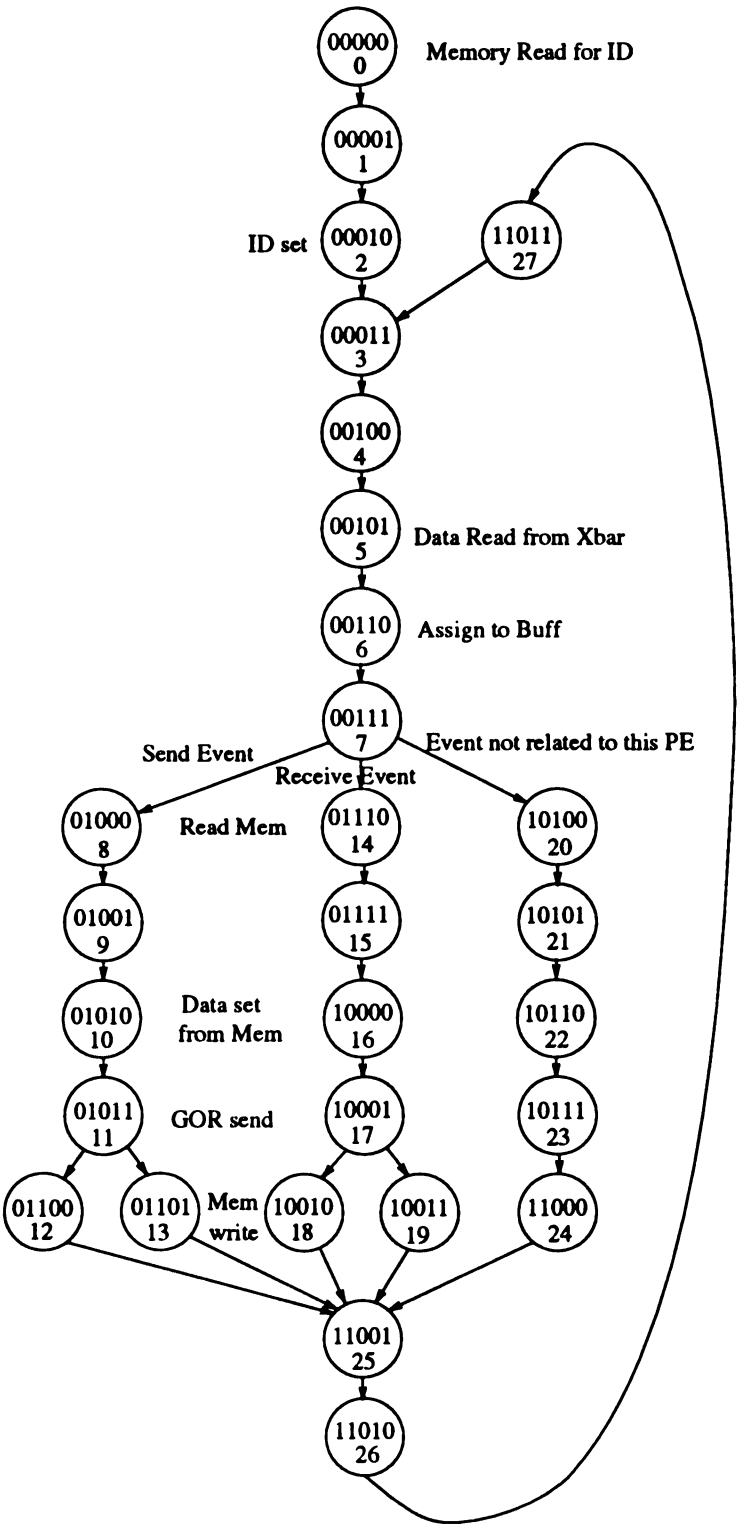


Figure 5.1: State Transition Diagram for the Splash 2 Implementation of On-The-Fly Order Algorithm for PE 1 to 16

utilizes more logic than the other PEs.

The actual execution latency of the hardware subsystem implementing the on-the-fly algorithm can be found from the number of states for each iteration of the algorithm. As shown in Table 5.1, every PE goes through 13 states during each iteration of the algorithm. A transition occurs at every rising edge of the clock. Since the feasible operating clock rate is 17.9 MHz (i.e., 56 nsec period), the latency per iteration of the algorithm is $56nsec * 13 = 728nsec$. One iteration of the algorithm that returns one output (valid or NULL event record) takes the same amount of time, regardless of the number of states.

Some PEs stay in waiting states several times. These states are needed either for waiting for the completion of *Read Operation* or for synchronization with other PEs. The actual code of these states is the same, except for their state numbers. These waiting states can be implemented in two different ways: by using as many separate waiting states as possible or by using counters to stay in the same state for the required number of cycles. In the remainder of this chapter, different modeling approaches and their synthesis results are summarized. Three different modeling approaches, (1) using Crossbar connection of Splash 2 to broadcast data and no counter is used, (2) using Crossbar and with counter, and (3) using Serial connection instead of using crossbar, are described below:

- Approach 1: Using Crossbar without Counter

  This approach utilizes the crossbar connection of Splash 2. Many duplicate waiting states, e.g., states 3, 4, and 20 through 27 in Figure 5.1, are used in

```
PROCESS
  BEGIN
    WAIT ON clk;
    IF (present_state = "10100" AND clk = '1') THEN
                transition <= "10101";     -- state 21
    ELSE
                transition <= NULL;
    END IF;    -- end of state 20

END PROCESS;

PROCESS
  BEGIN
    WAIT ON clk;
    IF (present_state = "10101" AND clk = '1') THEN
                transition <= "10110";     -- state 22
    ELSE
                transition <= NULL;
    END IF;    -- end of state 21
END PROCESS;

-- ...
```

Figure 5.2: Sample VHDL Code for a Waiting States Without Counter (Approach 1)

this approach. Input data are broadcasted to all PEs via the crossbar. Thus, each datum is processed in 13 cycles. All PEs examine the same input data concurrently. Sample waiting states are shown in Figure 5.2. The result of this approach is summarized in Table 5.1. The corresponding synthesis results for PE0 and the rest of the PEs are shown in Figure 5.3 and Figure 5.4, respectively.

- Approach 2: Using Crossbar with Counter

  This is a modification of approach 1. Here, the VHDL model uses an integer counter to repeat the waiting states. Counters, which are mapped to adders,

Maximum clock speed: 18.9 MHz (52.8ns)
Number of states: 21
Number of cycles needed: 13

	No. Used	Max Available	% Used
Occupied CLBs	293	400	73%
Packed CLBs	139	400	34%
Bonded I/O Pins:	115	160	71%
F and G Function Generators:	105	800	13%
H Function Generators:	26	400	6%
CLB Flip Flops:	279	800	34%
IOB Input Flip Flops:	0	160	0%
IOB Output Flip Flops:	0	160	0%
Memory Write Controls:	0	400	0%
3-State Buffers:	0	880	0%
3-State Half Longlines:	0	80	0%
Edge Decode Inputs:	0	240	0%
Edge Decode Half Longlines:	0	32	0%

Figure 5.3: Approach 1 Synthesis Results of PE0 for Broadcasting Data using X-Bar

Maximum clock speed: 17.9 MHz (56.0ns)
Number of states: 28
Number of cycles needed: 13

	No. Used	Max Available	% Used
Occupied CLBs	200	400	50%
Packed CLBs	95	400	23%
Bonded I/O Pins:	82	160	51%
F and G Function Generators:	80	800	10%
H Function Generators:	18	400	4%
CLB Flip Flops:	191	800	23%
IOB Input Flip Flops:	0	160	0%
IOB Output Flip Flops:	13	160	8%
Memory Write Controls:	0	400	0%
3-State Buffers:	0	880	0%
3-State Half Longlines:	0	80	0%
Edge Decode Inputs:	0	240	0%
Edge Decode Half Longlines:	0	32	0%

Figure 5.4: Approach 1 Synthesis Results (PEi) for Broadcasting Data using X-Bar

```
PROCESS
VARIABLE counter : INTEGER := 0;
  BEGIN
    WAIT ON clk;
    IF (present_state = "10100" AND clk = '1') THEN
        IF counter = 5 then
                XP_Mem_RD_L <= '1';
                XP_Mem_WR_L <= '1';
             -- ... other operations
                transition <= "11001";     -- go to state 25
        ELSE
                counter := counter + 1;
                transition <= "10100";     -- same state
        END IF;
    ELSE
                transition <= NULL;
    END IF;    -- end of state 20
END PROCESS;
```

Figure 5.5: Sample VHDL Code for a Waiting State Using Counter (Approach 2)

are used to reduce the total number of waiting states. Although the number of states can be reduced to 21 (compare to approach 1, which requires 28 states), the synthesis result from this approach operates a slightly more slowly than the previous approach and occupies more space because of adders and comparators (50% vs. 58% CLB occupancy rate). The portion of the VHDL code used in this approach is shown in Figure 5.5; synthesis results are shown in Figure 5.6.

- Approach 3: Using Splash 2's Serial Connection

  In Splash 2, the system can be used for systolic application by using its left-to-right serial connections. In this approach, a crossbar connection is not used; instead, PEs are connected serially. Input data are fed from PE1, and after 8

Maximum clock speed: 15.0 MHz (66.5ns)
Number of states: 21
Number of cycles needed: 13

	No. Used	Max Available	% Used
Occupied CLBs	233	400	58%
Packed CLBs	94	400	23%
Bonded I/O Pins:	82	160	51%
F and G Function Generators:	188	800	23%
H Function Generators:	55	400	13%
CLB Flip Flops:	172	800	21%
IOB Input Flip Flops:	0	160	0%
IOB Output Flip Flops:	1	160	0%
Memory Write Controls:	0	400	0%
3-State Buffers:	0	880	0%
3-State Half Longlines:	0	80	0%
Edge Decode Inputs:	0	240	0%
Edge Decode Half Longlines:	0	32	0%

Figure 5.6: Approach 2 Synthesis Result for PEi for Broadcasting Data using X-Bar

```
Maximum clock speed:   11.2 MHz (89.4ns)
Number of states: 15
Number of cycles needed: 8*16
```

	No. Used	Max Available	% Used
Occupied CLBs	304	400	76%
Packed CLBs	120	400	30%
Bonded I/O Pins:	118	160	73%
F and G Function Generators:	241	800	30%
H Function Generators:	82	400	20%
CLB Flip Flops:	232	800	29%
IOB Input Flip Flops:	0	160	0%
IOB Output Flip Flops:	13	160	8%
Memory Write Controls:	0	400	0%
3-State Buffers:	0	880	0%
3-State Half Longlines:	0	80	0%
Edge Decode Inputs:	0	240	0%
Edge Decode Half Longlines:	0	32	0%

Figure 5.7: Approach 3 Synthesis Results (PEi) for Systolic Modeling Approach

cycles each datum is passed to the right neighbor. An integer counter is used to reduce the number of waiting states. The synthesis results of this approach are shown in Figure 5.7.

Chapter 6

Discussion

In Part II, modeling is shown to be an important first step in the design process. The modeling techniques described in the previous chapters are used to implement such system components, and these models are used in high-level synthesis process.

The importance of modeling issues has been demonstrated in Part II. Several modeling techniques of FSM and the implementing FSM based on VHDL modeling are presented in Chapter 2. Different modeling styles affect not only design reusability and simulation results but also the synthesized results. In particular, a rapid prototype of a signal convolution application has been developed, and it is synthesized and realized using Xilinx FPGAs for the Splash 2 system. It is described in Chapter 4. Another example of modeling styles affecting synthesized results is illustrated in Chapter 5 with the on-the-fly sorting algorithm in an Instrumentation System.

VHDL has been shown to be suitable to model a system component, parallel architectures, and parallel algorithms. Several parallel architectures and algorithms are modeled using VHDL and these are described in Chapter 2 and 3.

Part III

CAD Framework

Computer-Aided Design Frameworks are design environments consisting of design tools that aid design activities. The CAD framework's support for the design process has three parts: *specification, execution,* and *services. Specification* corresponds to how tasks can be decomposed, what tools are available, and how they may be used. *Execution* is concerned with what methodology or process to select for a given task, what tool to invoke, and how to invoke it. *Services* support the coordination of subprocesses and enforce consistency. Although the concept of the CAD framework can be applied to many different engineering design disciplines, the discussion here is concerned with high-level synthesis only.

Part III describes a proposed *execution environment* for a CAD framework highlighting the selection and execution of design methodologies by system guidance, as well as the separation of the proposed execution environment from design process specification, unique to this system.

Part III is organized as follows: In Chapter 7, existing CAD frameworks are reviewed and their strengths and weaknesses summarized. In Chapter 8, high level synthesis processes are discussed. Chapter 9 presents an overview of methodology management and summarizes design process specification. The proposed CAD framework's execution environment is presented in Chapter 10. Finally, in Chapter 11, the synthesis example using the implemented CAD framework is illustrated.

Chapter 7

Previous Work

Research efforts in design methodology management have been made as part of the CAD framework. One approach to finding a methodology is to utilize information about individual tools and to search for a sequence that accomplishes the overall task. The *Design Planning Engine*, which is an expert system, is used to build a design plan. The Design Planning Engine of the ADAM system [6, 7, 8] produces a *plan graph* using a forward chaining approach. Acceptable methodologies are specified by listing pre-conditions and post-conditions for each tool in a lisp-like language. Also, estimation programs are used to guide the chaining. The shortcomings of the ADAM system are that it is not amenable to simultaneous exploration of multiple alternatives, and it is difficult to modify with predictable results.

Another system with a similar approach is Chippe [43, 44, 45], in which Brewer *et al.* proposed the concept of *"knobs and gauges."* The evaluator generates design quality measures (gauges) such as area, delay, performance, component usage frequency, and others. The planner then uses these measures to set the design style

and strategies (knobs), such as connectivity style, pipelining level, and optimization order. A *design critic* uses the "knobs and gauges" paradigm to reach design goals. In this approach, iterative design procedures are applied at the design hardware component level only, since the design process is fixed. Iteration is done by changing design parameters and/or constraints. The iterative application of different design methodologies is not supported.

Ulysses [46, 47] and Cadweld [41, 48] use blackboard systems to control design processes. A knowledge source, which encapsulates each tool, views the information on the blackboard and determines when the tool would be appropriate. Ulysses [46, 47] uses blackboard architecture to construct the methodology. In CADWELD [41, 48], task requests are posted to a blackboard; CAD Tool Objects (CTOs) then volunteer, and the requester chooses among them. Here, CTOs describe tool applicability and invocation mechanisms. This system may also have the same shortcomings as stated above.

MINERVA [49] and the OCT task manager [9] use hierarchical strategies for planning the design process. Hierarchical planning strategies take advantage of knowledge about performing complex tasks involving several subtasks. Hierarchical planning, as used in MINERVA [49], is more effective because knowledge about how to perform logical subtasks can be applied more naturally. However, a severe limitation of this system is that it only allows a manual mode: every selection decision must be made by the designer.

NELSIS [50, 51, 52] provides *Flowmaps* for representing sets of methodologies. Flowmaps can be exponentially large if there are many alternatives. In the NELSIS

system, the flow plan is fixed. Rules are used to invoke a particular tool. A procedural language in which tasks are defined is used. Since tasks are executed manually by designers in the sequence they choose, this system does not possess the ability to define flow dynamically. Furthermore, no parallelism is allowed.

Kleinfeldt *et al.* review some important concepts in the design process management area and give an extensive overview of the existing systems [53].

Recently, Martin-Marietta and Lockheed have developed CAD frameworks to support the design process of Rapid Prototyping of Application Specific Signal Processors (RASSP) [54, 55, 56]. In both systems, a workflow shows possible ways a task can be carried out. Alternative methodologies are represented by *OR* nodes in a workflow. During the execution time, one of the nodes joined by *OR* is selected. Therefore, the actual execution and the specification of design methodology are not separate. To specify methodologies, Lockheed uses its own internal representation. In the RASSP Design Environment developed at Martin-Mariette, *Integration DEFinition (IDEF)* language [57, 58] is used to represent a workflow, and a dataflow graph is generated from the workflow to show the dependency between the tasks. IDEF includes both a definition of a graphical modeling language (syntax and semantics) and a description of a comprehensive methodology for developing models. Both systems are targeted mainly for environments in which methodologies are fixed. Both systems have several shortcomings:

- It is not easy to incorporate process metrics in selecting methodologies.

  Methodologies must be selected based on the trade-off determined by process

metrics in an ideal system. Examples of the process metric entries are reliability, development time, manufacturing cost, life cycle, system defects, software/hardware reuse, inter-operability, physical constraints such as volume and weight, and power constraints. These process metrics are usually known as the design process proceeds and cannot be determined in advance. However, in both approaches, workflow is determined in advance before actual design starts.

- Representing dependencies between design data and methodologies is not possible.

  Methodologies selected during the design process depend on what data are available and how previous tasks have been carried out. For example, when there are multiple tools for the same tasks which are not interchangeable at the lowest level, representing this situation is problematic in either approach. Sometimes, one tool or methodology selected in the previous step may affect the tools to be selected later. Since dependency cannot be represented in either system, the CAD framework cannot help the designer to select methodologies which satisfy such constraints.

- Systematic search of the design space is very difficult.

  The CAD framework should help the designer select the design methodology that best satisfies the requirements. If the chosen methodology is not satisfactory, alternative methodologies must be pursued. Also, the designer should be able to investigate multiple alternatives simultaneously (with different design parameters or with different versions). However, the lack of formalism in both

approaches makes it difficult for the CAD frameworks to search the design space efficiently.

In [59, 60], a formal approach to design methodology management has been proposed. In contrast to many other approaches, this new approach separates the specifications of methodologies and the execution environment. In [59, 60], the process grammar is used to define all possible methodologies. The details of how methodologies are selected and executed are left to the designers. Designers must supply manager programs to select and execute methodologies.

Several research efforts have used high-level Petri Nets [11] to model the execution environment of software systems. DECOR [61] and HILDA [62] each use an extension of Predicate Transition Nets to specify a design process, such as tool invocation sequences and their relations. In DECOR, design process monitoring and control is realized by interpretation of the net. Hierarchical places and transitions are used to decompose the net into several subnets.

In HILDA, production rules are used to select alternatives, and these rules are attached to the Petri Net. Static dependencies between tools, scheduling decisions, conflicts, exception handling and backtracking, and tool parameter selection are examples of knowledge types for making rules. Alternative selections are made by the user, making it difficult to modify rules after backtracking.

A system called Monitor has been implemented for CAD tools using a Petri Net [10]. The Monitor uses fixed design methodologies. This system has been very limited in terms of system functionalities and services. Monitor's design flow graph

is represented as a fixed Petri Net, and the design flow follows this graph. Some of the main features of the system are (1) graphical display of tools, files, and their relationships, (2) automatic invocation of selected tools and report of termination status, and (3) the tracking of the overall system state.

Development of a Colored Petri Net (CPN) is described in [63], in which CPN is used to speed up the design and validation of VLSI chips at the register transfer level. The chip design process is broken down into several pipe-lined stages and each stage is modeled as a transition in CPN. A similar system can be found in [64], where the task of modeling and validating the behavior of a VLSI chip using Hierarchical CPN is presented.

Related works related to design data management can be found extensively in the literature. In [65, 66], Katz *et al.* described their data model *Version Server*, which provides a complete range of services needed to control the versions and configurations of a complex design as it evolves over time. Checking in and checking out design objects, selecting the preferred design version, and binding dynamic configurations are some of the system's operations.

Kim elaborated further on configuration and version problems in the CAD design environment in his dissertation [2]. He adopted the object-oriented concept and applied this concept to his hierarchical data model. This model was applied to VLSI design data such as VHDL design environments.

For distributed design environments, managing design data is a more challenging problem since design data have various relations and are shared by several users or several tools at the same time.

By managing persistent data and run time data, the creation of a new version is detected and saved in the persistent database in a distributed design environment [67]. In order to support the handling of concurrent accesses, a *Server-Client* scheme is used to arbitrate the accesses and control versions of design data in the distributed environment. A server is a process which handles persistent design data and arbitrates requests from many CAD tools, each of which is a client. Any change is notified by the server to all the clients who need to use the new data. In [68], Schettler *et al.* also presented a data model for the persistent design data storage and manipulation in a CAD framework.

Design flow management is another field which has been concentrated upon by many researchers in CAD frameworks. Earlier efforts used a fixed design flow. For example, Monitor used a predefined design flow which was represented as a Petri Net [10]. *Flowmap* was used to control the design flow [51]. Similar approaches can be found in [69]. In their work, Miyazaki *et al.* used a *flow-chart* based design task flow graph. Most of these flow-based approaches used a fixed design flow, which lacks flexibility and the capability to dynamically construct design flow.

To overcome such limitations, a CAD framework should support the dynamic construction of design flow. A *task graph*, a directed acyclic graph with each node in the graph corresponding to an entity in the *task schema* and each edge corresponding to a dependency, is generated dynamically [70]. In their work, Sutton *et al.* used the task schema which specifies the dependencies between design entities, both tools and data. Based on the task schema (the dependency between tools and data) designers

can *expand* the abstract node in the task graph or *execute* a tool if the node is a primitive task.

Chapter 8

High Level Synthesis

In this chapter, the steps involved in the high-level synthesis are briefly explained. The proposed CAD framework helps designers in selecting the right tools and right methods in each of these steps. High-level synthesis takes as inputs a high-level description or an abstract behavioral specification of a hardware behavior and a set of constraints, and produces, as a result, a structural description which implements the behavior satisfying the goals and constraints. The output of high-level synthesis is usually a register transfer level (RTL) description. Many steps are involved in high-level synthesis. Typical high level synthesis steps are shown inside of the broken lined box in Figure 8.1.

The first step is usually the compilation of the formal languages into internal representations (*Flow Graph Generation*). The next steps comprise the scheduling of operations, functional unit allocation, and controller generation. These steps are represented as *Architectural Synthesizer* in Figure 8.1. In the compilation step, most approaches use variations of graphs containing both the data and control flow implied

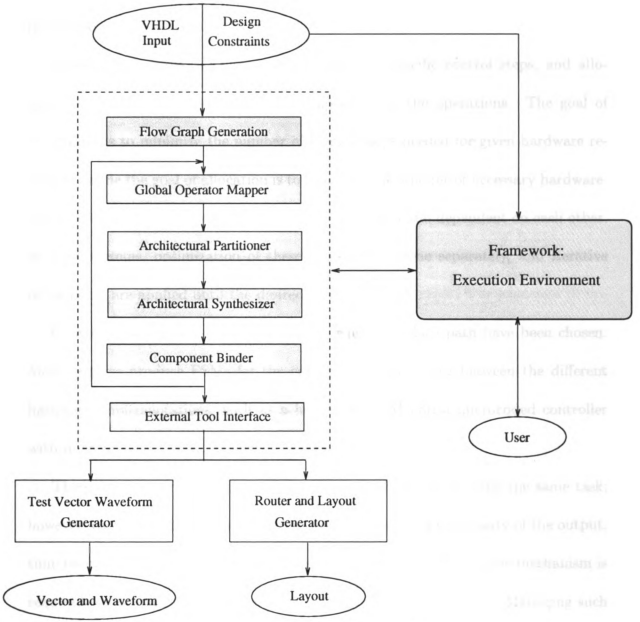


Figure 8.1: High Level Synthesis Steps

by the behavioral specification. The data flow graph shows the essential ordering of operations in the behavioral specification. At this step, some initial optimizations, such as dead-code elimination, inline expansion of procedures, and loop unrolling, are performed.

Scheduling involves assigning the operations to specific control steps, and allocation is actually assigning hardware components to the operations. The goal of scheduling is to minimize the number of control steps needed for given hardware resources, while the goal of allocation is to minimize the amount of necessary hardware. Thus, scheduling and allocation are closely interrelated and dependent on each other. In most systems, optimization of these two steps is done separately, and iterative refinements are applied until the desired goal is reached.

Controller synthesis is done after the schedule and data path have been chosen. Most systems produce FSMs for the controllers. The choice between the different hardware implementations, such as a hardwired FSM and a microcoded controller with microprogram steps, influences the final product.

There are many existing tools or programs capable of performing the same task; however, each tool may produce different results in terms of the quality of the output, time requirements for the task, resource requirements, etc. Thus, some mechanism is required to optimize tool selection in terms of type, time, and order. Managing such a task is one of the most important roles in a CAD framework.

Chapter 9

Methodology Management

Methodology is a set of processes or approaches used to solve a given problem. Methodology management is a technique employed to control these processes or approaches so that a better solution can be found. In the CAD area, design methodology management provides "the definition, presentation, execution, and control of design methodologies in a flexible, configurable way" [53]. The goals of design methodology management are to help the designer reduce the design time and to produce a better design.

In the last couple of decades, there has been a change in trends in the CAD community. The main focus of the CAD framework has shifted from managing data and tools to managing the design process itself.

Design methodology management should provide specification methods, an execution environment, and miscellaneous services. To choose appropriate specification methods, the following questions must be answered: How can tasks be decomposed? What tools are available? How will they be used? In the execution environment,

the management system must provide a means of selecting the appropriate design methodology or process for a given task and must determine the choice and method of tool invocation. Miscellaneous services, such as graphical user interface for communicating between the user and system, supporting cooperating subprocesses in the system, and enforcing consistent designs, should also be included in the execution environment.

The basic building blocks of a design methodology management system or a CAD framework are tools. In general, a tool cannot be decomposed into any subcomponents; thus, the CAD framework has no way to break a tool down into smaller tools. Each tool performs a specific function. A design methodology management system determines how to use these tools as well as when to use them. The sequence of tool usage is viewed as a design flow.

A large electronic design has the following characteristics:

- Hierarchical Design: A large design can itself be hierarchically organized, and the same is true for the design process. The whole design process can be broken down into several steps of subprocesses, where each subprocess can again be decomposed into multiple sub-subprocesses. A large system design can be partitioned into smaller functional subcomponents. Each of these subcomponents can be composed of other components. Multiple design teams can work together to produce this design.

- Multiple Design Representations: A design process can be viewed as a series of data transformations from one representation to another. Each transformation

produces a different type of design data. Furthermore, the same design transformation can be used with different sets of constraints and design parameters. Consequently, each of these transformations creates a different version of the data.

- Large Design Space: Many alternative processes for a task create many different versions of data, just as different values of design parameters lead to different results. As the size of design data increases, the time required to search the design database increases as well. Thus, a large design space should be maintained such that an efficient way of searching through the database is possible.

- Large Number of Tools: Many tools are involved in the system design process. These tools should be well maintained so that the right tool is used at the right time.

These characteristics lead to the use of an integrated design environment or CAD framework that can manage such issues effectively and guide designers to produce a viable designs.

In [5], a CAD framework is defined as "a software infrastructure that provides a common operating environment for CAD tools." In order for such a software infrastructure to provide a good integrated design environment, the CAD framework should support the following services: *Design data management, Design methodology management, Tool integration/encapsulation,* and *User interface* [71]:

- *Design data management*: Design data management deals with the methods used to store and retrieve design data as well as maintain relationships (such

as version, transformation, and configuration) and consistencies between designs. In the CAD design process, many different data files (either different intermediate results or different versions from the same task) must be stored and retrieved as needed. Thus, an efficient way of managing design data is necessary. This also includes version and configuration management. Design data management assists in the use of technology-independent design data. If the data is technology-independent, these data can be reused in different design processes without changing them.

- *Design methodology management*: This should provide a formal representation method of the design process and a seamless way of carrying out the design process. Design methodology management selects the best tool for a given input and constraints, and guides the designer to produce the best design. In other words, design methodology management is responsible for selecting and executing an appropriate sequence of tools to produce a desired design adhering to the given specifications. Thus, it guides the user in selecting the right tools in the correct order. The CAD framework should also support concurrent engineering concepts. For these reasons, design methodology management has gained a lot of attention in the past couple of decades.

- *Tool integration*: CAD tools can be integrated into the design environment by using a well-defined tool integration method. In order to handle many different tools which accept different types of input and produce different types of output, the CAD framework must provide an inter-tool communication mechanism.

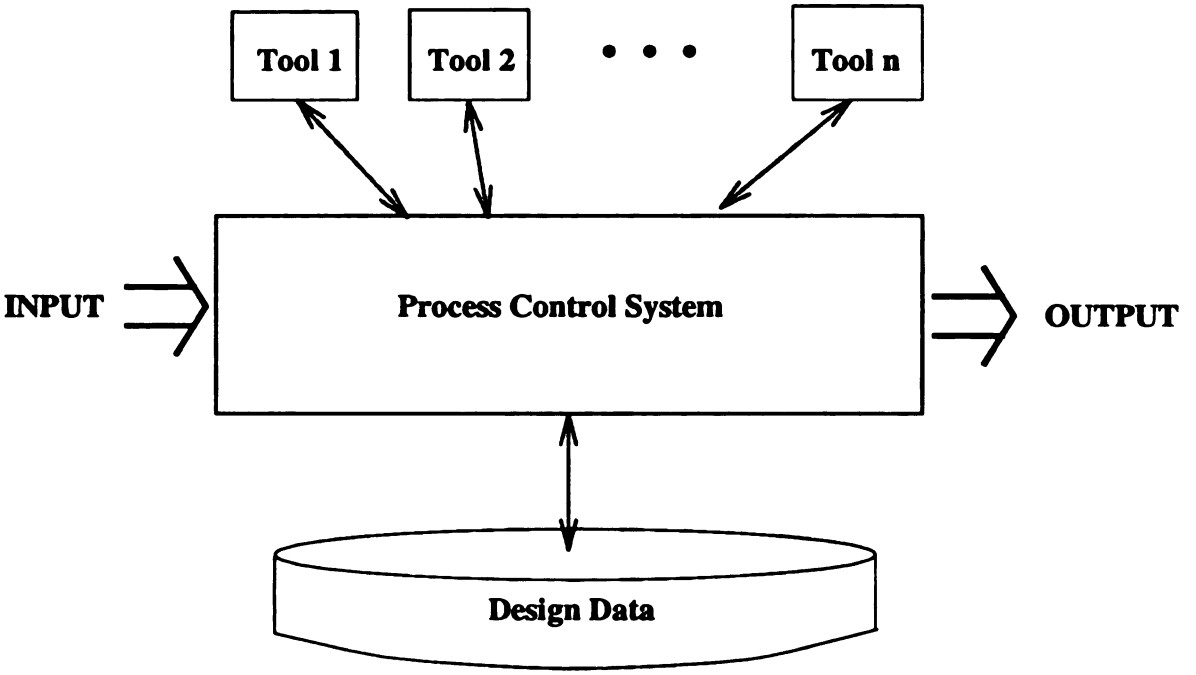


Figure 9.1: Block Diagram of CAD Framework

This facility ensures that all tools in the system can communicate with each other.

- *User interface*: The user interface should be easy to use and effective. It should also hide low-level implementation details as much as possible from the end user (e.g., tool-invoking sequences and commands).

The block diagram of such a CAD framework is shown in Figure 9.1. The CAD framework helps the designer to reduce design time and errors to produce a better solution.

9.1 Design Process Specification

Managing the design process necessitates the specification of the design process. The design process consists of a series of transformations, from input to output specifications. A *process flow graph* describes which tools and/or subtasks are used to transform input into output specifications, and shows the tasks as well as the information flow of the design process. A task can be decomposed into several subtasks, which may in turn be recursively decomposed into several sub-subtasks. For a given task, a process flow graph (called a *workflow* in other literature) shows the decomposition and the dependency between subtasks and design data. This hierarchical decomposition of processes enables a system to capture the design process efficiently.

A *graph grammar*, also called a *process grammar* [72, 73], provides a convenient means of transforming process flow graphs into progressively more detailed process flow graphs. The user indicates the goals of the design exercise by supplying a start graph, which indicates what input specifications are available, what output specifications are desired, and what logical tasks are to be performed. This graph is progressively modified by applying *productions* of the process grammar. Logical task nodes are replaced by subgraphs composed of smaller logical tasks and intermediate specifications.

9.1.1 Process Flow Graphs

Process flow graphs describe the information flow of a design process. Process flow graphs describe which tools or subtasks transform input into output specifications.

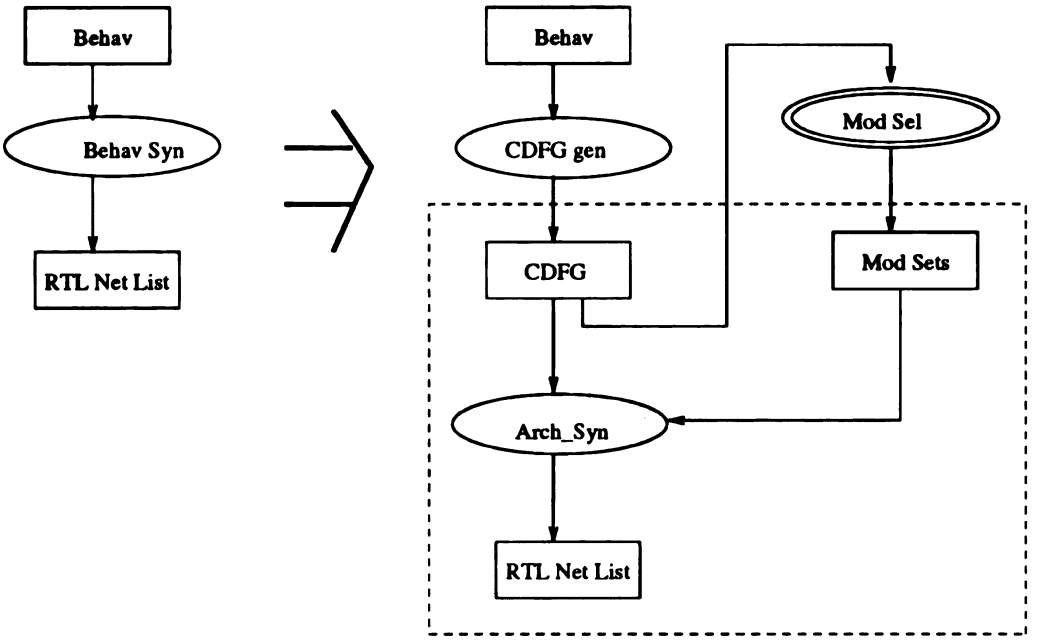


Figure 9.2: Production Graph Example

Formally, a process flow graph is a bipartite acyclic directed graph of the form $G = (T, S, E)$, where T is the set of task nodes (drawn as a ellipse in Figure 9.2), S is the set of specification nodes (drawn as a rectangle in Figure 9.2), and E is the set of edges indicating which specifications are used and produced by each task.

There are two types of task nodes: *terminal* and *logical*. A terminal task node represents the execution of an application program, commonly called a *tool invocation*. Non-terminal task nodes represent logical tasks, which could potentially be completed by using several different tools or a combination thereof. A logical task node is represented by a single-line ellipse, while a terminal task node is represented by a double-line ellipse. Process flow graphs can describe design processes with varying levels of detail. A graph containing many logical task nodes indicates what should be done, in abstract terms, without describing exactly which tools should be used.

Conversely, a graph in which all nodes are terminal describes a methodology completely.

9.1.2 Design Process Grammars

Graph grammars provide a convenient means for transforming process flow graphs into more detailed process flow graphs. Baldwin and Chung defined *design process grammar* [72]. Productions using a design process grammar permit the replacement of one subgraph by another. A production in a design process grammar can be expressed as a tuple $P = (G_{LHS}, G_{RHS}, \sigma_{in}, \sigma_{out})$, where

G_{LHS}, G_{RHS} are process flow graphs for the left and right side of the production, respectively, such that G_{LHS} is a single, logical task node representing the logical task to be replaced.

σ_{in} is a mapping from input specifications of G_{RHS} to input specifications of G_{LHS}. Each input specification must be mapped to one with the same type or subtype.

σ_{out} is a mapping of output specifications from the left hand side of the production to the right hand side of the production. Each output specification must be mapped to one with the same type or subtype.

A design process can be specified using design process grammar. In the proposed CAD framework, the specification of the design process is separated from design flow management, a feature that is absent from all other existing CAD frameworks.

Figure 9.2 illustrates some productions for the Behavioral Synthesis task. As shown in this figure, the logical design task *BehavSyn* is carried out by control

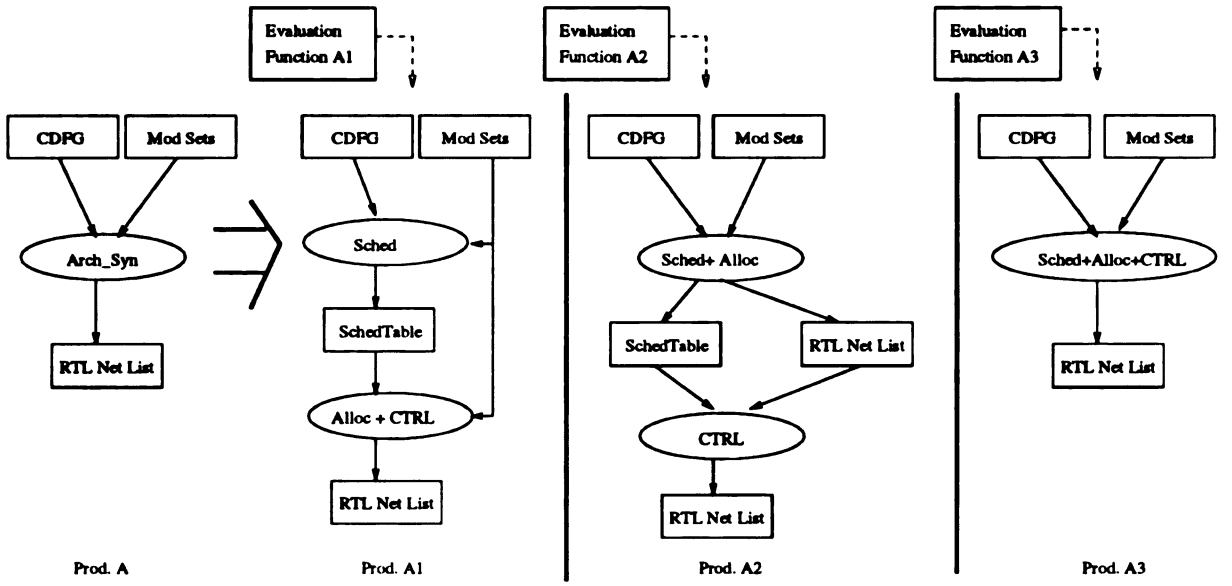


Figure 9.3: Production Graph Example 2

data flow graph (CDFG) generation *CDFGgen*, followed by module selection *ModSel* and the architectural synthesis *Arch_Syn*. Each of these tasks, such as *BehavSyn* or *Arch_Syn*, may be another logical or terminal task which invokes a tool.

The logical task *Arch_Syn*, which represents the architectural synthesis process, can be carried out by one of three alternatives: *Prod. A1*, *Prod. A2*, or *Prod. A3* shown in Figure 9.3. The vertical bar is a shorthand notation indicating multiple productions with the same G_{LHS} but different G_{RHS}'s. In this example, alternative 1, alternative 2, and alternative 3 explain how *Scheduling, Allocation,* and *Controller Synthesis* can be carried out.

In the production graph 9.2, the task *Arch_Syn*, enclosed by a broken rectangle, is replaced by the selected production if alternative 2 (*Prod. A2* in Figure 9.3) is applied. An expansion example of *Arch_Syn* is shown in Figure 9.4. Process flow graphs in general are discussed in more detail in [72, 73].

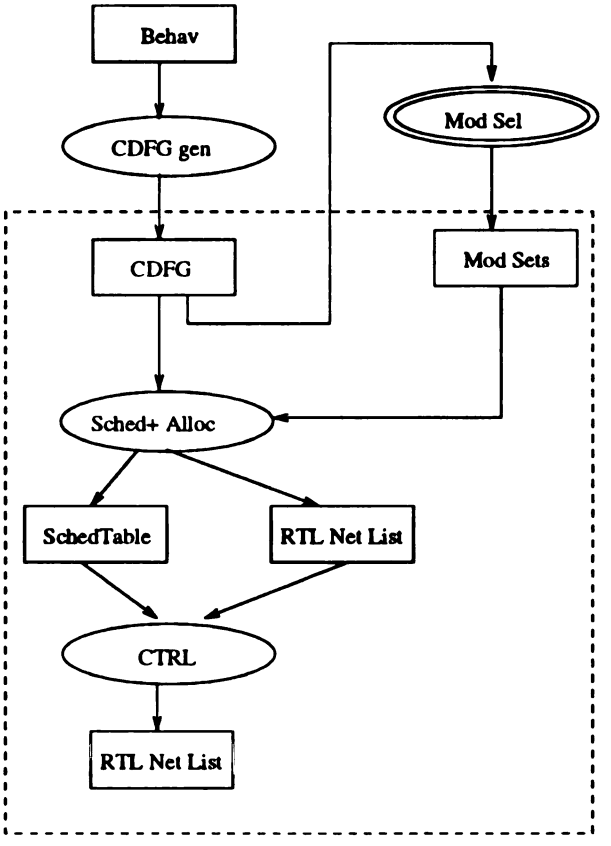


Figure 9.4: Expansion Example

9.1.3 Specification Hierarchy

Task specifications are defined and organized into a specialization and generalization class hierarchy. Properties of general task specifications are also available in a special task specification where the general task specification is a parent of the special task specification. A child specification inherits its parent's specification properties. For example, pre- and post-evaluation conditions can be inherited by children; however, any child specification can have its own condition through specialization.

It is possible to decompose tasks hierarchically into simpler tasks until the individual tasks can be performed by single tool invocations. Methodologies are devised by hierarchically decomposing logical tasks until all tasks are terminal. A single tool selection can be considered to be a special case of decomposition in which the set of subtasks is a single terminal task. A specification node definition editor window is shown in Figure 9.5. An example of task hierarchy, the ***Layout Synthesis Task***, is shown in Figure 9.6. Among many layout design styles, three common layout design styles are *Gate Array, Standard Cell,* and *Full Custom.* These design styles can be applied to any level of layout synthesis hierarchy. These style conditions can be passed on to children tasks in the hierarchy. However, ***MCM Layout*** should have different conditions to be imposed than the conditions for ***PCB layout*** and ***Chip Layout*** because in MCM layout, inter-chip delay can be ignored.

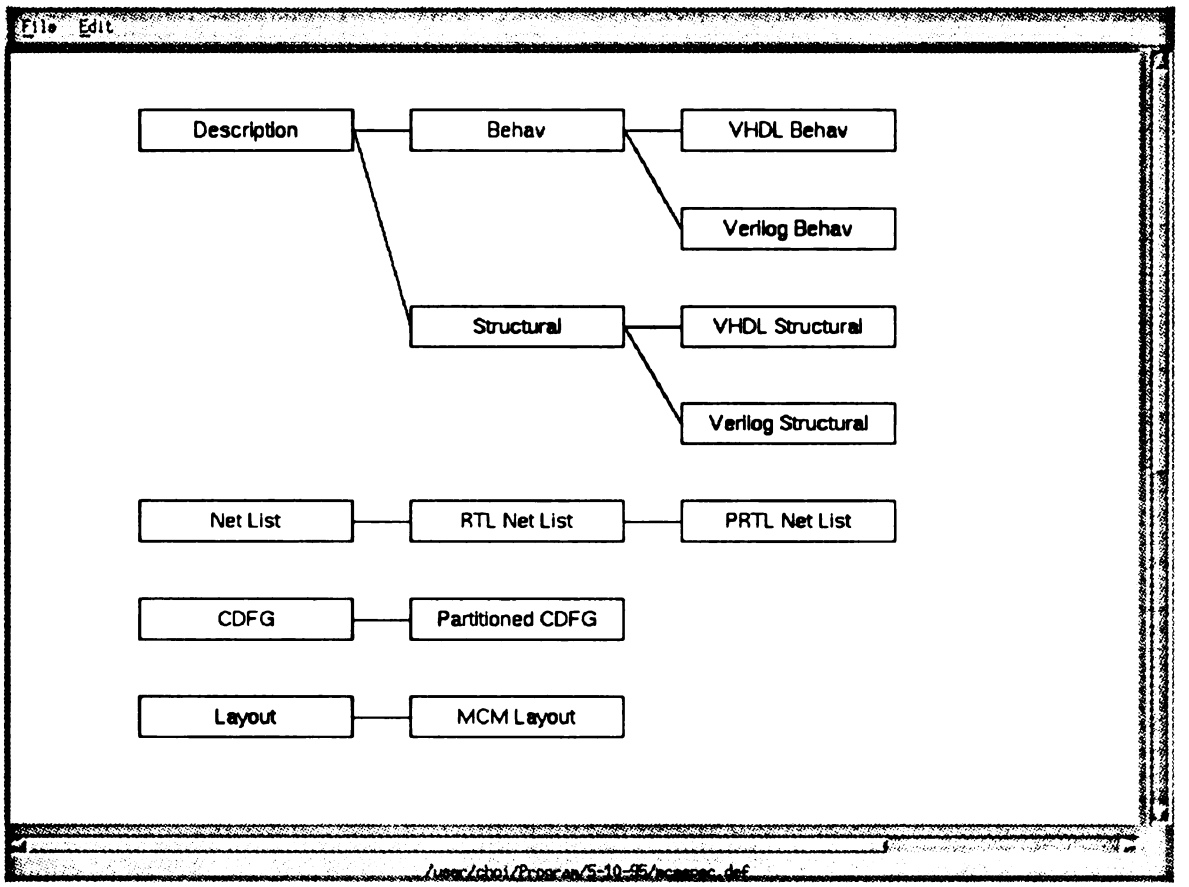


Figure 9.5: Specification Definition Editor Window

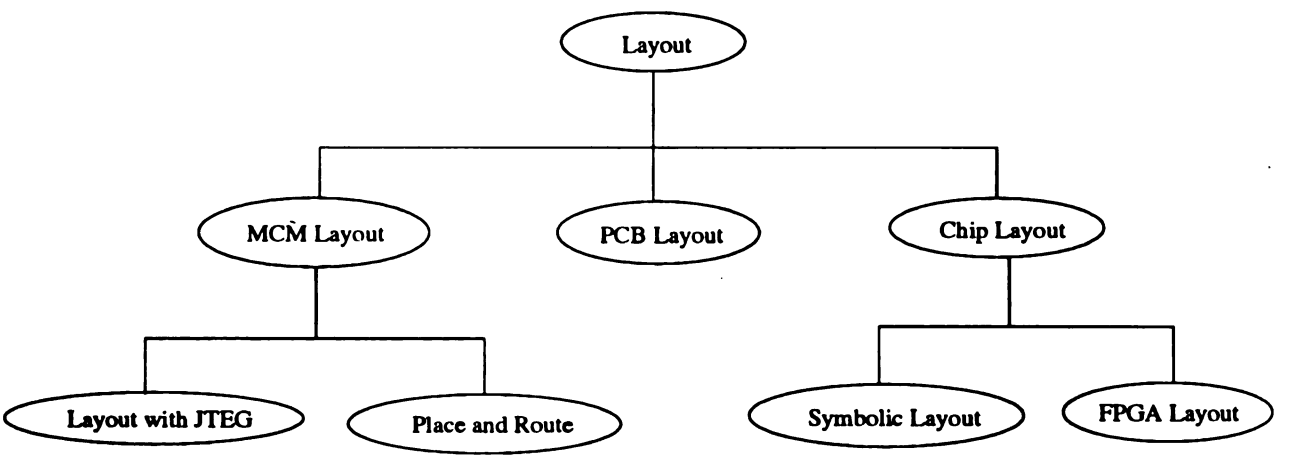


Figure 9.6: Layout Synthesis Task Hierarchy

9.2 Execution Environment

The CAD framework execution environment is a software environment which helps designers in selecting and executing design methodologies by allowing the systematic exploration of the design planning space. The execution model within the execution environment is modeled based on Petri Net. When inputs are available, tasks are executed and outputs are created.

The execution environment allows backtracking, which occurs when the task cannot be accomplished. If the execution environment detects unsatisfactory results, the system is allowed to go back and try different alternatives. Another advantage of the execution environment is that it allows parallel exploration of the design space. Details of the execution environment are covered in Chapter 10.

9.3 Tool Encapsulation

Tool integrations can be challenging because every tool vendor has its own batch of command line options and environment variables which must be correctly set. Tool invoker programs that invoke corresponding tools must set the environment variables and command line options appropriately. Electronic CAD vendors and users have recognized the importance of inter-operable tools and created the CAD Frameworks Initiative (CFI) to facilitate the development of CAD frameworks which integrate tools. CFI helps to address this problem, suggesting standards for data interchange, inter-tool communication, and tool encapsulation [74, 75, 76, 77]. CFI proposed a Lisp-like Tool Encapsulation Specification (TES) language providing a uniform format

for vendors to specify how to set environment variables, compose command lines, and interpret exit codes. CFI standards do not govern the selection of tools.

CFI tries to achieve inter-operability between different tool vendors by proposing standards, such as the Intertool Communication standard, the Message Dictionary Specification standard, and TES. There are several issues associated with tool encapsulation. First of all, the current TES proposal may be changed as CFI further modifies its requirements and methods. Another difficulty is that tool vendors do not yet provide TES files. Finally, inter-operability between different tools is very problematic to maintain in terms of communication since each tool may produce a different form of an intermediate file. If there were a standard intermediate file, then at most $O(n)$ translations instead of $O(n^2)$ translations would be necessary. This is one reason why standards are recommendable. For example, an EDIF standard format were used in low level design representation, and any output at this level could be translated into an EDIF format.

Chapter 10

The CAD Framework: Execution Environment

In this chapter, the execution environment is discussed. The execution environment of the proposed CAD framework is modeled based on Petri Nets. A new approach to the execution environment, which dynamically constructs a process graph, automatically selects design alternatives, and automatically backtracks if the result is not satisfactory, is presented in this chapter. This chapter is organized as follows: First an overview of the proposed CAD framework architecture is given in Section 10.1. Petri Nets are reviewed in Section 10.2. The formal model of the execution model is explained in Section 10.3. The backtracking mechanism is explained in detail in Section 10.4, and in Section 10.5, the handling of multiple alternatives is explained. Other issues, such as constraints, process simulation, and version control, are covered in subsequent sections.

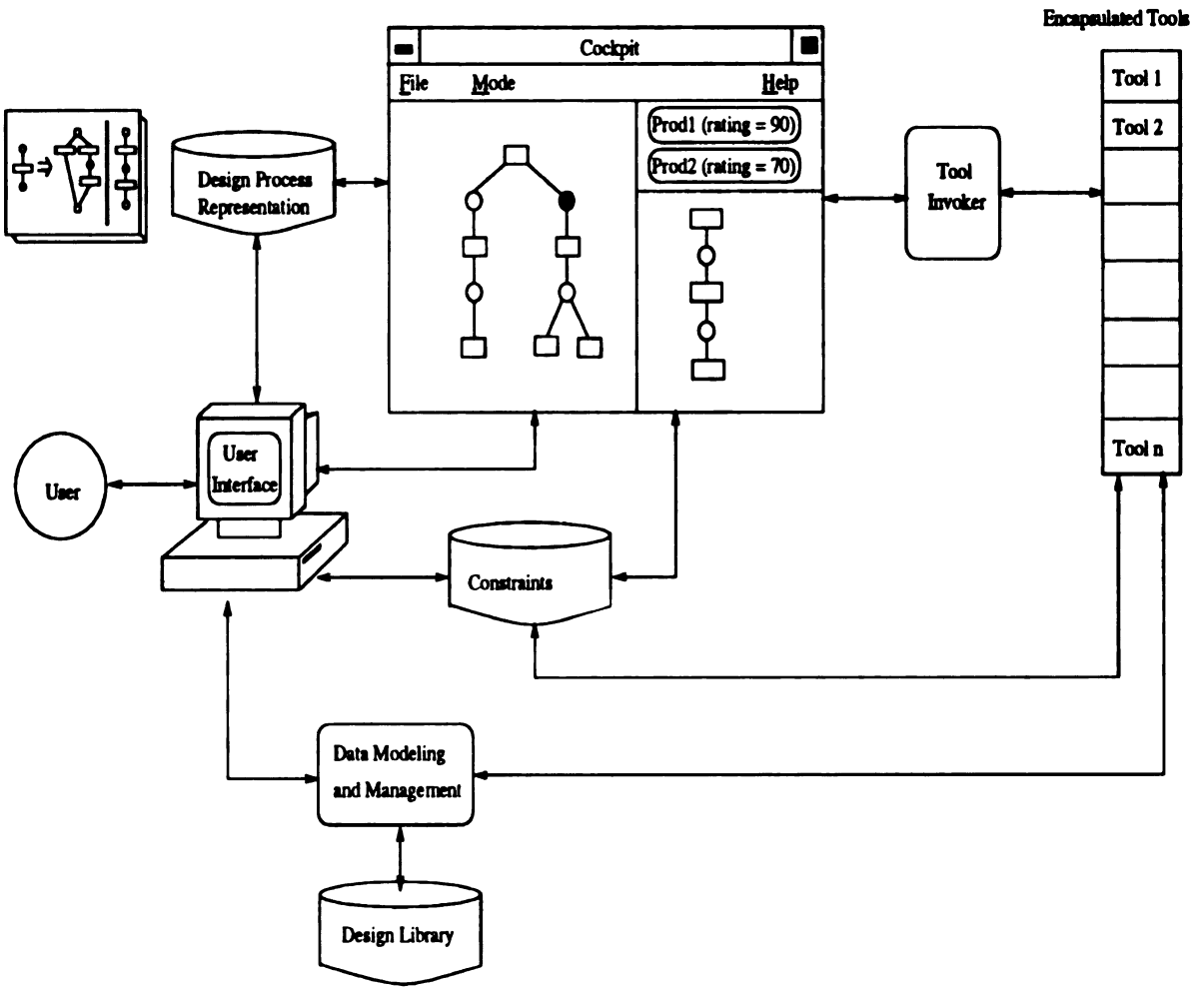


Figure 10.1: Block Diagram of System

10.1 Proposed CAD Framework Overview

The proposed system is composed of several main components: *Design Process Representation*, *Constraints*, a *Design Library*, an *Execution Environment or Cockpit*, and *Graphical User Interface*. An overview of the proposed architecture of the CAD framework is graphically depicted in Figure 10.1. Design Process Representation represents design methodologies using the productions of a process grammar. Productions codify the possible hierarchical decomposition of tasks, which designers use to build a process flow graph. The process grammar naturally captures the hierar-

chical character of the design process and allows systematic exploration of the design space.

Design constraints are provided by the user. The system performs a pre-evaluation in order to select the best production or tool, and also performs a post-evaluation after a task is finished. When pre-/post-evaluation processes are carried out, the system uses the constraints as input parameters. Constraints are items such as area, critical delay, die size, pin number, power consumption, etc. The Design Library contains various design data.

The execution environment program, Cockpit, keeps track of the design status and communicates with the designer via the Graphical User Interface (GUI). The GUI helps users in several ways. Users can browse the available productions via the GUI and assign one or more input design data files together with their control information. The design progress can be displayed either in the form of a production or a Petri Net structure. The design path is displayed if the user chooses the history menu. Design data examination and a display of scoring results are additional features of the GUI.

10.2 Petri Net

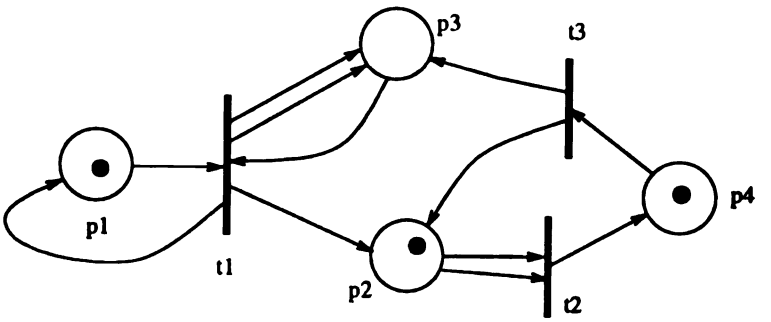
Petri Net theory was developed by Carl Petri in 1962 to model a system by using a mathematical representation of the system [11, 12, 13, 78]. Petri Nets offer a means to graphically and mathematically model discrete event systems. Moreover, Petri Net models can be converted into computer control mechanisms that can be interfaced to discrete event handling processes such as high-level synthesis processes. As a graphical

tool, Petri Nets can be used as visual communication aids similar to flowgraphs.

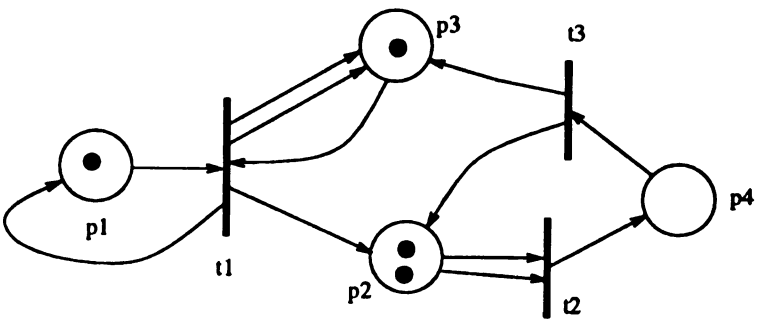
DEFINITION 1 *A* Petri Net Structure, C, *is a four-tuple,* $C = (P, T, I, O)$. $P = p_1, p_2, ..., p_n$ *is a finite set of* **places**, $n \geq 0$. $T = t_1, t_2, ..., t_m$ *is a finite set of* **transitions**, $m \geq 0$. *The set of places and the set of transitions are disjoint,* $P \cap T = \emptyset$. $I : T \rightarrow P^\infty$ *is the* **input** *function, a mapping from transitions to bags of places.* $O : T \rightarrow P^\infty$ *is the* **output** *function, a mapping from transitions to bags of places [13].*

The *execution* of a Petri Net is handled and controlled by the number and distribution of its *tokens.* Tokens are primitive concepts of Petri Nets and have no inherent meaning. Tokens are assigned to the places of Petri Nets. A transition is *enabled* when there are tokens in each of the transition's input places. A transition is ready to be fired if all of its input places have tokens in them. A Petri Net executes by *firing* enabled transitions. A transition fires by removing tokens from its input places and creating new tokens which are distributed to its output places. After a task finishes its execution, the input tokens are removed from input places and new tokens are produced and placed in the output places.

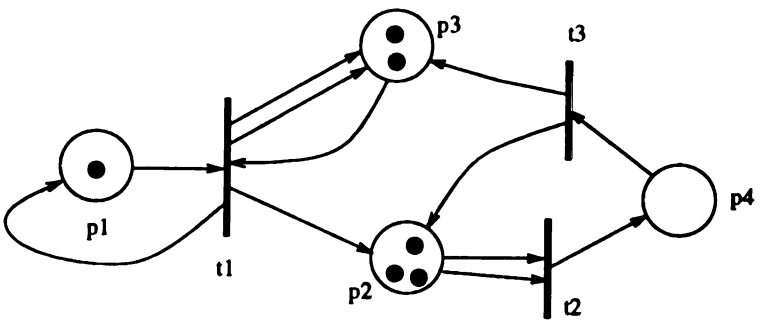
Sequences of firing enabled transitions in an ordinary Petri Net are nondeterministic. An example sequence of firing transitions in an ordinary Petri Net is shown in Figure 10.2. In this figure, Graph A shows an initial graph with initial token distributions. At this moment, transition *t3* is the only one which is enabled since its input place, *p4*, has one token. After transition *t3* is fired, a new graph is created; this is shown in Graph B. A new token is added in both *p2* and *p3*. Now, two transitions, *t1* and *t2*, are enabled. In this case, the order of firing transitions *t1* and *t2* is unde-



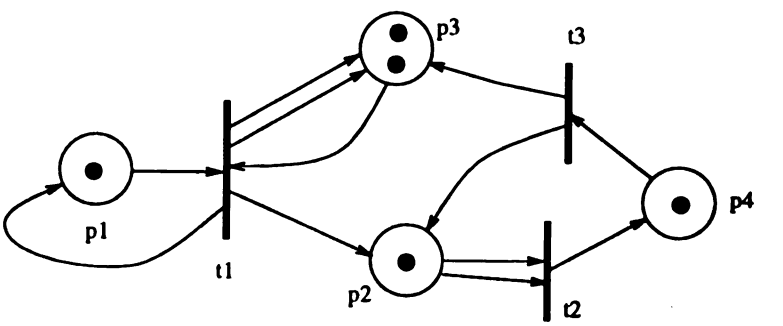
(A) Initial Graph



(B) After t3 is fired



(C) After t1 is fired



(D) After t2 is fired

Figure 10.2: Petri Net Firing Sequences

termined. Either transition can be fired first, both may also be fired concurrently. If the transition *t1* is fired first, then Graph C becomes the new graph after transition *t1* is fired. The same situation, where both *t1* and *t2* are enabled, again occurs. At this time, if the transition *t2* is fired, then Graph D results. The execution continues until there are no more transitions which can be fired.

Many different system models and applications have used Petri Net. Modeling of hardware, communication protocols, parallel programs, and distributed data bases are several examples of such applications [11, 78, 79, 80]. Different extensions have been made for modeling of different systems which have avoided its ordinary non-deterministic behavior and allowed the extended semantic meanings of the ordinary Petri Net. Jensen [63] uses an extended Petri Net to design VLSI chips. In his extension, transitions correspond to hardware components, places hold design data, and tokens correspond design data. Firing of transitions cause the execution of the hardware components using input data. Di Janni [10] describes similar extensions for the VLSI circuit design system, Monitor. In Monitor, transitions are CAD tools and tool invocations correspond firing of the transitions. In addition to these extensions, conditional firing of transitions, inhibitors, are also used. Kljaich et al. use an extended Petri Net called *flow nets* to describe and verify the fault-tolerance capabilities of digital systems [81]. In their work, additional extensions are made: symbolic token and user-supplied *types*. Token has a symbolic value, such as data and control information, associated with it. Different types are associated with transitions, and places. Different output tokens are generated depending on the transition type after the transition is applied.

There are several benefits to using a Petri Net model for the execution environment. The Petri Net model captures design processes naturally. Designers can see the design progress easily. Moreover, by manipulating tokens, execution control is much easier than that of ad hoc methods. For example, if the designer assigns three tokens in an input place of a production, three alternatives will be carried out.

The execution model proposed in this thesis is based on an *Colored Petri Net.* The proposed execution model described in this thesis does not make any extension to Colored Petri Nets, but it has adopted several extensions which had been made by various people. Contribution of the execution model described in this thesis is the application of Colored Petri Net for the process control execution model.

The extensions adopted by the proposed execution model will be described in the subsequent paragraphs. First of all, in the proposed execution model, logical and terminal tasks are modeled as transitions, and design data are modeled as tokens. Tokens contain several data fields, such as a design data name list field, a version history field, token color, and other control fields. Thus, tokens have both data and control information.

In the proposed model, tokens are *not consumed*, but their states are changed as transitions occur. In ordinary Petri Nets, tokens are consumed after transitions are fired. In [63, 64], data values attached to tokens are referred to as token colors. The values of token colors are predefined in a color set. In the proposed model, token color represents the states of token, such as used, not yet used, or used but failed. Thus, in this Colored Petri Net, semantic meanings are assigned to tokens by their colors. Semantic meanings of token colors are explained in Section 10.3.4.

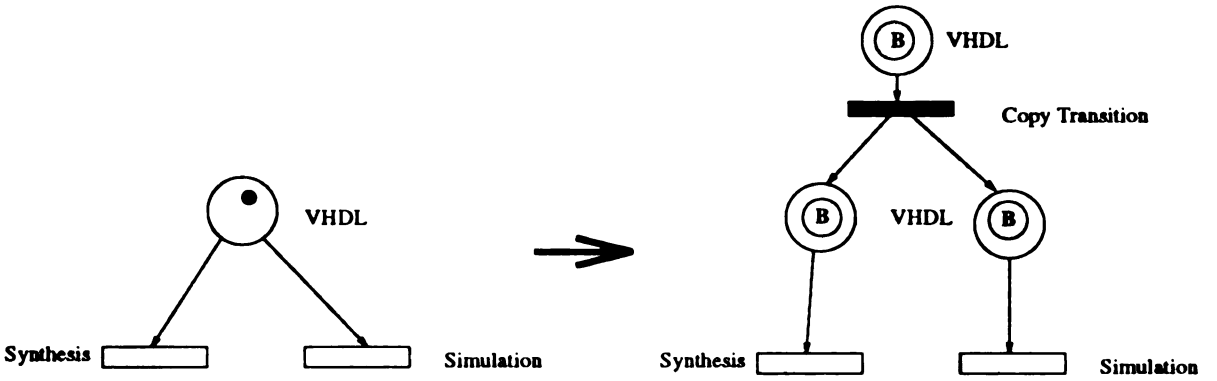


Figure 10.3: Copy Transition

Two additional transitions are introduced, *pre-evaluation* and *post-evaluation* transitions in this model. These transitions are not a part of the actual design process. After finishing a pre-evaluation transition, the system puts the different-colored output tokens in the corresponding input places, depending on the pre-evaluation function result. This transition makes the execution environment deterministic and helps to select suitable production alternatives. After post-evaluation transitions, the output tokens are generated *conditionally* depending on the result of the evaluation. If the post-evaluation fails (e.g., the result is not satisfactory), the token color of the starting production is changed in order to allow backtracking. If the transition succeeds, the new result is placed in the normal output place.

Another transition is needed when an input datum is used in more than one transition. For a production alternative, only one transition is selected and a single input token is used for the transition. However, when there are several transitions in a production which use the same token, the token needs to be copied and distributed to the these transitions. The token is duplicated by introducing a new transition named *Copy Transition* shown in Figure 10.3.

10.3 Execution Model

The execution model provides for dynamic execution of tasks and the representation of state information. At a minimum, the execution model allows the designer to access tasks and designs by tracking the information required to invoke tasks. The execution model constructs a process graph by selecting the proper production for each logical task. This selection is guided by invoking a pre-evaluation of the alternatives of the logical task. When an appropriate task is selected, the execution model either expands the graph or invokes a tool for the execution of a terminal task. After executing the tool, the execution model post-evaluates the result based on the criteria (constraints). If the result is not satisfactory, it backtracks to try another alternative.

The execution model is based on an *Colored Petri Net* and performs the following functions:

- Dynamic Construction of a Process Graph
- Pre-Evaluation of the Alternatives
- Selection of a Production for each Logical Task
- Execution of a Tool
- Post-Evaluation of the Result
- Backtracking if Needed

The execution model creates design flows by reading the production graph, determines the possible design alternative processes, and invokes the right tool for exe-

cution or expands the logical task. In order to choose the right alternative, Cockpit performs a *pre-evaluation* of all available logical tasks. An evaluation function is associated with each logical task. When the pre-/post-evaluation process is carried out, Cockpit uses the constraints as one of the input parameters.

Expansion is dynamically performed as the design process progresses. When the production graph is read by Cockpit and is converted into a corresponding Petri Net internal structure, a pre-evaluation function is called for each alternative and the results are posted, such as the score of each alternative, in the net. The highest score enables a corresponding transition. The scheduler in Cockpit now schedules or chooses which transition is to be fired based on resource availability. After finishing one path, the result is checked; this is called post-evaluation. If this does not agree with the anticipated result, the system backtracks to the selection point and tries another alternative.

10.3.1 Cockpit

Cockpit is a routine of the execution environment which performs the following functions:

- Creating Daemon processes (Initial Graph)
- Keeping track of the design process,
- Dynamically constructing a process graph,
- Scheduling task(s) by pre-evaluation,

- Performing the post-evaluation,
- Interacting with the user, and
- Controlling the GUI.

Cockpit is implemented using the algorithm described in Figure 10.4. Cockpit initially creates several Daemon processes which maintain task specific knowledge. Cockpit's information about the design process comes entirely from an input file indicating a set of possible tasks and those decompositions that should be considered for each logical task.

The user interacts with Cockpit, which keeps track of the current status of the design process and informs the user of possible actions. Cockpit's display indicates to the user what design tasks have been completed so far and what tasks remain.

To assist the user in choosing an appropriate action, Cockpit invokes several evaluation functions. The evaluation functions provide ratings for the possible task decompositions and check the results. The ratings help the system to select tools. Cockpit determines what decompositions are available for the remaining logical tasks. This information is then displayed to the user.

Cockpit supports two modes of operation: manual and automatic. The manual mode is normally used for high-level decisions and stepping through the design process. The designer may wish to use the automatic mode for lower level decisions. In the manual mode, Cockpit waits for the user to select a decomposition or *execute* a task. In this mode, the system performs pre-/post-evaluation and the system guides the user by showing the results. The user makes the final decision for selection of

```
Initialization()
  {
    Start graph is selected;
    Create initial Daemon process and place tokens (send message);
  }
Wait for message;
   IF the message is from Execution process THEN
     {
       IF the message is WANT_EXPAND THEN
        {
          Invoke Pre-Evaluation function;
          Select Production based on Pre-Evaluation;
          Display expanded process flow;
          Send EXPAND_THIS or FAILED message to Execution process;
        }
       IF the message is POST_EVAL THEN
         {
            Post-evaluation;
            Send result POST_EVAL_OK or _FAIL to Execution process;
         }
       IF the message is FAIL_EXPANSION THEN
          Delete useless tokens;
       IF the message type is FAILED THEN
          Kill the child process;
     }
   ELSE /*  Message from Daemon process */
     {
       IF the message is FAILED THEN
            Kill the child process;
       ELSE
          {
            Create a Daemon for the subsequent task;
            Put the output token in the newly created Daemon's input
              place;
          }
     }
   END IF;
```

Figure 10.4: Algorithm for Cockpit

a production or backtracking based on the suggestions made by the system. When the user selects a decomposition, Cockpit displays the new subtasks in place of the original task. When the user requests that a task to be executed, Cockpit sends a message to the corresponding Daemon process for execution. For terminal tasks, the tool invoker responds by invoking a tool. The user invokes the automatic mode by executing a logical task instead of selecting a decomposition. In response to an execution message for a logical task, the Daemon process uses encoded knowledge from a process graph to select a decomposition and then executes the subtasks (also in automatic mode). If necessary, the designer may reverse any decision made by the Daemon process in the manual mode.

In the automatic mode, the execution model utilizes the Petri Net structure more naturally since there is no human interaction. Cockpit dynamically creates design flows by reading the production graph, determining possible design alternative processes and invoking the correct tool for execution or expanding the logical task. In the execution environment, Cockpit uses the evaluation function to determine the alternative. After finishing the task execution, Cockpit *post-evaluates* the result.

When the output of a task is not satisfactory, it is necessary to backtrack. Either different parameters must be supplied to some of the tools or different tools must be chosen; or alternatively, the task must be decomposed in an entirely different way. The designer may request that certain task decompositions be reversed. Additionally, if a decomposition was requested by a Daemon process, that process can direct Cockpit to reverse it. Cockpit saves the state of the session before backtracking in case the designer later decides to cancel the reversal.

10.3.2 Daemon Processes

Each Daemon process is invoked (created) by Cockpit and Execution process. Daemon processes are dynamic repositories of task-specific knowledge. Each message from the Daemon process indicates the task being evaluated or executed and provides all the inputs and outputs file names. The constraints may be included in one of the input files or may be passed to the Daemon process directly.

Each Daemon process is activated by an event signaling the arrival of a token in its input or output places. If an input event occurs, the Daemon process creates an execution process by sending the task name to Cockpit to create the process. For an output event, the Daemon process checks the output token numbers, which is assigned by the user. If the number of tokens does not reached the required number, the Daemon process tries a yet untried alternative by changing the input token color. The usage of colored tokens is explained in detail later in Section 10.3.4. If more than enough tokens are generated, the Daemon process selects the best tokens. Each Daemon process retains the following information:

1. Parent Task Name
2. Name of Input Places
3. Name of Output Places
4. Child Task Names
5. Information about the required input and output token numbers (counters)

The procedure for Daemon process is shown in Figure 10.5.

```
Wait for messages;
IF the message is from its parent (Execution) process THEN
  {
    IF the message is TOKEN_IN THEN
       Create a child Execution process;
       (Send the production name to the child process)
  }
IF the message is from its child (Execution) process THEN
  {
    IF the message is FAILED THEN
       Send the FAILED message to its parent (Execution) process;
    IF the message is TOKEN_IN THEN
       Send the TOKEN_IN message to parent (Execution) process;
  }
```

Figure 10.5: Algorithm for Daemon Process

10.3.3 Execution Process

Each Execution process is created by Daemon and receives a production name. The execution process is responsible for invoking a tool, asking for expansion, and asking Cockpit to do a post-evaluation after finishing its job. *Each Execution process handles only one token at a time.* For multiple tokens, one Execution process is created for each token. The Execution process contains information about Input Places, Output Places, and its Task Name. The corresponding algorithm is shown in Figure 10.6.

10.3.4 Token Semantics Extension

Tokens have different semantic meanings from those in the regular Petri Net model. Tokens contain real design data as well as control information to control the design process. For example, control information is used by the system to choose appropriate

```
Execution()
  {
   IF Terminal task THEN
      Execution of the terminal tool;
      (Send TOKEN_IN message to parent Daemon process)
   IF Logical task THEN
      {
        Send WANT_EXPAND message to Cockpit;
        Wait reply from Cockpit;
          IF the message is EXPAND_THIS THEN
            {
               Expand the graph by creating one Daemon process for
                 each task node in the production;
               Send TOKEN_IN message to the child Daemon process;
               Wait for messages from these children Daemon processes;
               IF the message is from child Daemon process THEN
                 {
                    IF the message is TOKEN_IN THEN
                      {
                         Send POST_EVAL to Cockpit;
                         Wait for reply from Cockpit;
                         IF POST_EVAL_OK THEN
                            Send TOKEN_IN to parent Daemon process;
                         IF POST_EVAL_FAIL THEN
                           {
                              Send FAIL_EXPANSION, FAILED, WANT_EXPAND
                                 to Cockpit;
                           }
                      }
                    IF the message is FAILED THEN
                      {
                         Send message to parent Daemon process;
                         Exit;
                      }
                 }
            }
          IF message is FAILED THEN
             Send FAILED to parent Damon process and Exit;
      }
  }
```

Figure 10.6: Algorithm for Execution Process

alternatives.

Each token has one of several colors as control information. A color is assigned to a token to indicate the status of the design data and control transitions. As a transition is fired, the token is not consumed, but its color is changed, signifying a change of state. After finishing a task, the output is examined based on given constraints.

The token colors used in the system are as follows:

- BLUE Token: Each BLUE token indicates that the corresponding alternative must be tried and that the output token must be generated whether the output is acceptable or not.

- YELLOW Token: This token indicates that the corresponding alternative has not yet been tried. When backtracking occurs, a YELLOW token is changed to a BLUE token for an alternative trial.

- RED Token: If the output does not meet the constraints, the corresponding input token is changed to RED, indicating that this production has failed.

In Figure 10.7, alternative 1 with a BLUE token is applied. Here, alternative 2 with a RED token has already been applied and failed, while alternative 3 with a YELLOW token has not yet been applied.

Figure 10.9 illustrates the steps involved in the dynamic expansion of the Petri Net using the productions shown in Figure 10.8. Figure 10.9 (A) shows an initial graph with one Blue token in the input place of a transition, *HLS*. This token enables the transition, and since the task is logical, the graph is expanded as shown in Figure 10.9 (B). When a logical task is expanded, pre-evaluation and post-evaluation transitions

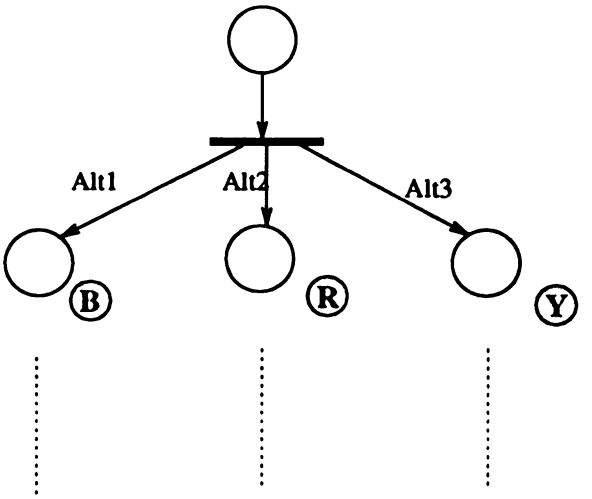


Figure 10.7: Token Color

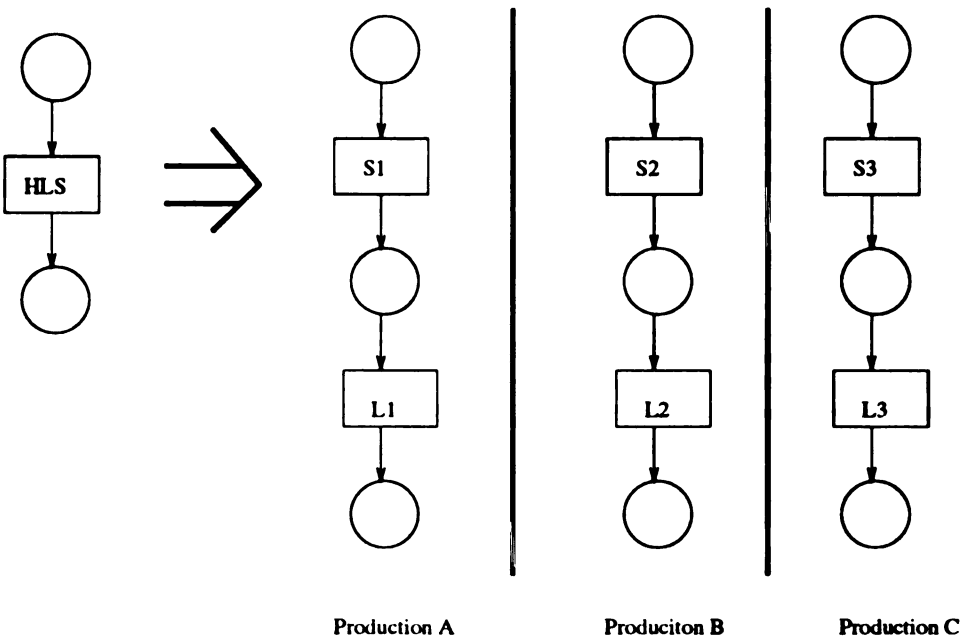


Figure 10.8: Production Example for Expansion Steps

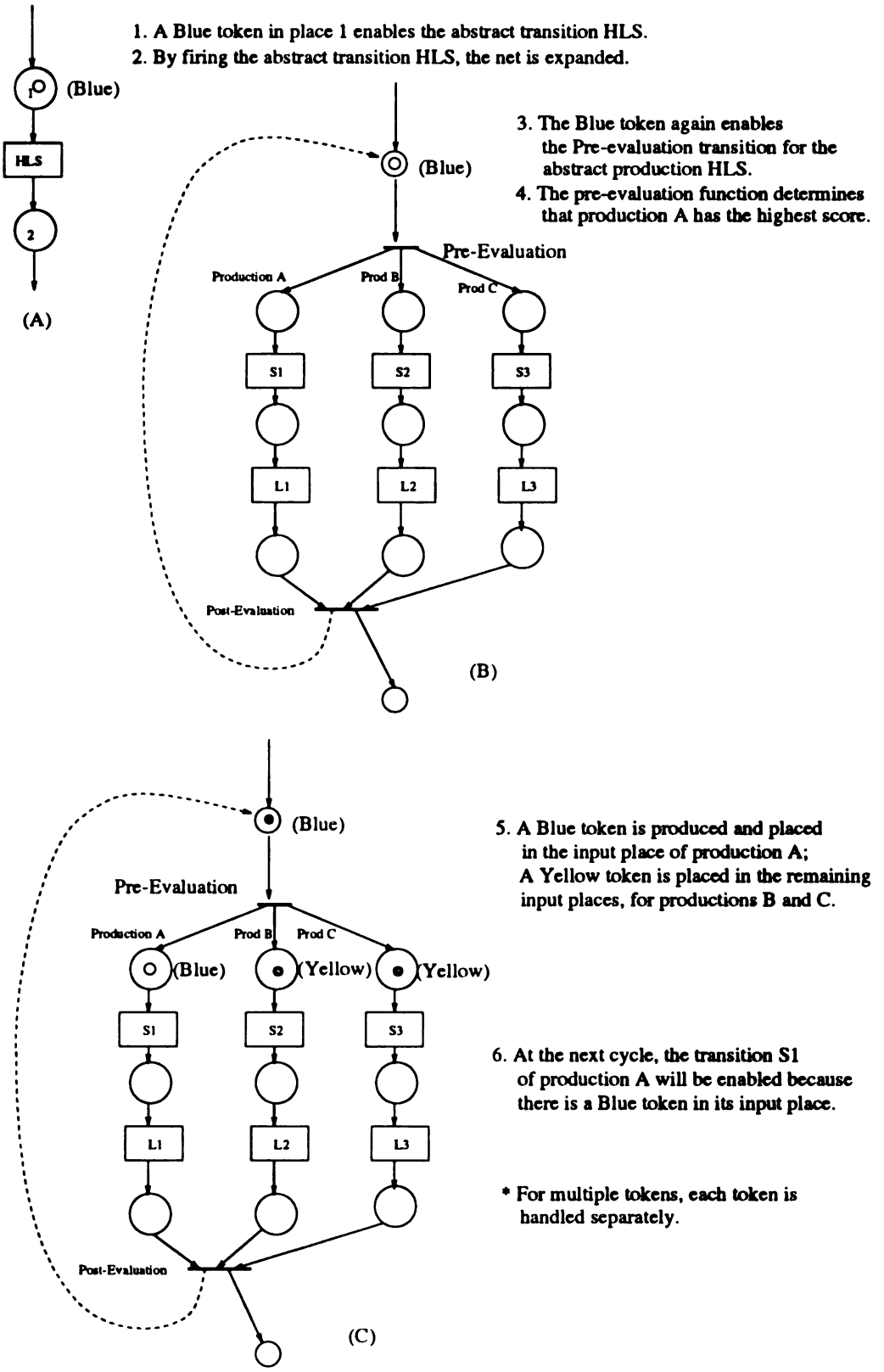


Figure 10.9: Expansion Steps

are added to the Petri Net. A *Backtracking Path* (shown in broken lines) is also added, as seen in Figure 10.9 (B and C). The Blue token enables a pre-evaluation transition. In Figure 10.9 (C), the pre-evaluation transition determines the best alternative by invoking the evaluation function. If *Production A* receives the best score, the transition places a new Blue token into its output place in order to pursue Production A as a selected alternative. The remaining alternatives receive Yellow tokens, and these tokens prevent the firing of these alternative transitions.

The user can limit the maximum number of concurrently executed alternatives through the number of tokens assigned to the production's input place. Likewise, the number of output tokens, which limits the number of output results, can be specified. The strategy for selection among multiple production outputs is also specified by the user. Currently available strategies are first-available (FA) or best-result (BC). Under *FA*, the post-evaluation transition selects the output first generated and ignores all other outputs; whereas under *BC*, all outputs are collected, compared and the best result(s) is selected. For example, in Figure 10.9 (c), assume that two productions, Production A and Production B, are concurrently applied. Under FA, the post-evaluation accepts whichever result was generated first from either Production A or B, while under BC, the post-evaluation postpones the comparison until both results are available.

10.4 Backtracking Mechanism

When the output of a task is found to be unsatisfactory during post-evaluation, it is necessary to backtrack. Different parameters must be supplied for some tools or different tools must be chosen; alternatively the task must be decomposed in an entirely different way. In the manual mode, the user controls the backtracking mechanism by directing the productions to be tried; however, in the automatic mode, Cockpit itself makes these selections based upon the given constraints.

Automatic backtracking is done by managing the token colors and traversing the backtrack path. Figure 10.10 (A) shows some logical tasks, and Figure 10.10 (B) illustrates the Colored Petri Net together with tokens when these tasks are fully expanded. This figure shows that the tasks *CDFG gen*, *Partitioning*, and *MCM Arch Syn* have 3, 2, and 2 production alternatives, respectively. For example, suppose that Path-3 is first selected by the pre-evaluation transition when CDFG-gen is executed. If the Post-evaluation-1 result is not satisfactory, the system tries to backtrack, but all the alternatives have already been tried, (in this case, there is only one path), and thus the resulting failure token (the Red token) is deposited in its output place, Place-1. Thus, in Figure 10.10 the system backtracks via the Back-2 loop and selects Path-2 for its next attempt. After finishing Path-2, Post-Evaluation-2 evaluates the new output from Path-2. If the result is satisfactory, the transition generates a successful token to its output place. This new token in turn enables the pre-evaluation for Path-4 and Path-5. The Yellow token in the unvisited path remains the same.

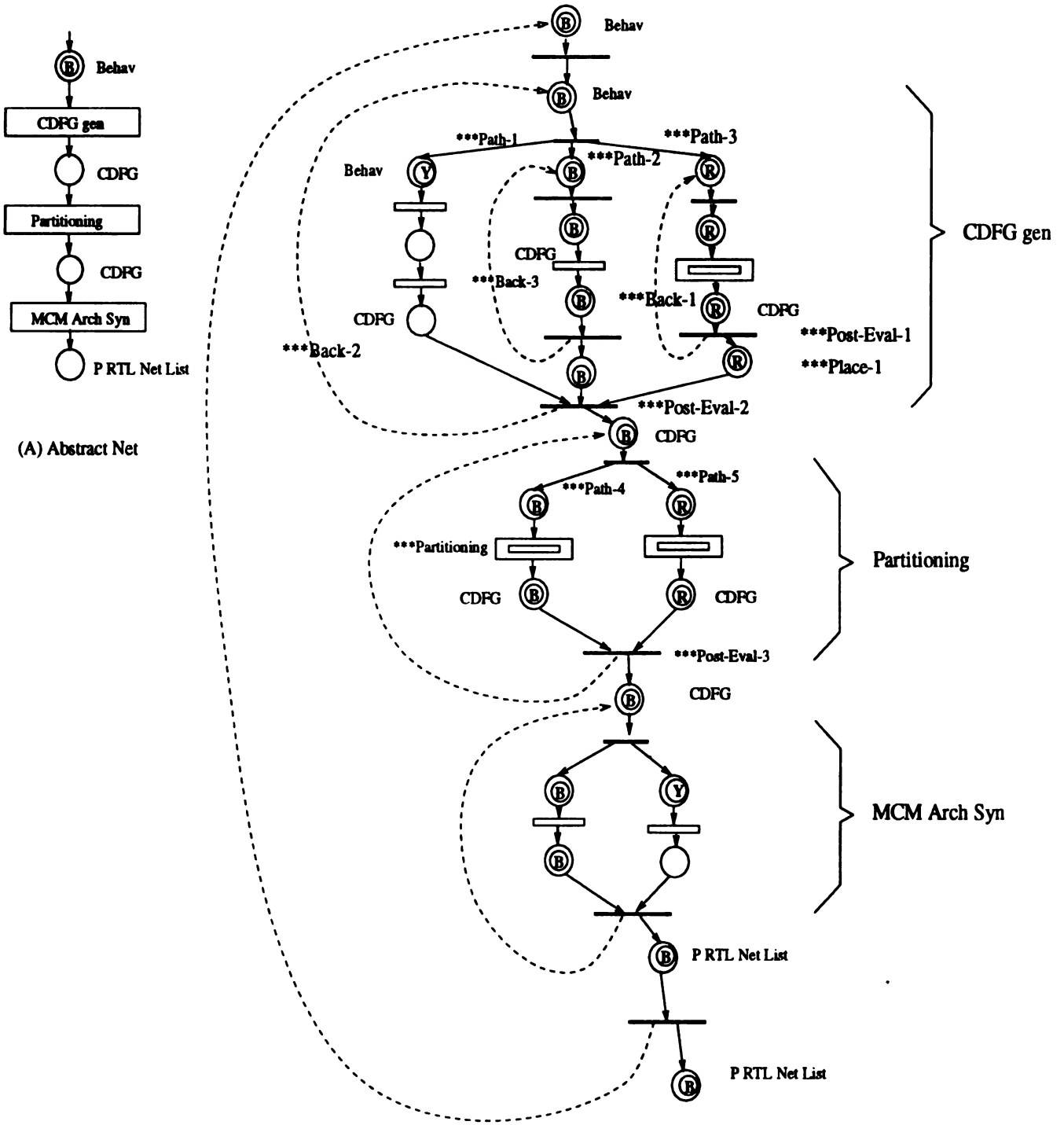


Figure 10.10: Backtracking Example

10.5 Multiple Alternatives

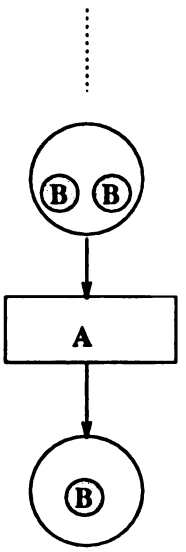
Several alternatives may be simultaneously explored. This helps the user to obtain better results by selecting the best solution among several solutions. There are two forms of parallel exploration of alternatives in the design process: use of multiple parameter alternatives and multiple production alternatives. For a given production, there may be several parameter choices available. If a production does not produce an output which meets the design constraints, the same production should be tried with different parameter sets until all possible parameter sets have been exhausted. In addition, a given logical task may be accomplished in several ways. Each methodology alternative represents a separate production for the logical task.

Multiple alternatives can be expanded and executed concurrently if the user specifies multiple tokens in the logical production's input place and/or output place via the GUI. The system assumes one token is in each place if the user does not place any token in the input or output places of a production. There are two cases which should be considered for multiple alternatives:

1. Multiple Tokens in the Input Place:

   The number of tokens indicates the number of productions to be simultaneously executed. If there is a child production which also has multiple tokens, it is carried out simultaneously. The total number of productions active at any given time is controlled by a global control variable, **Total_Production**, and by resource constraints.

2. Multiple Tokens in the Output Place:



2 Tokens for task A generates
only one output.
(Each input tokens generate output,
and among two outputs, one is selected.)

However, if tokens are used in other
parallel path nodes, then two tokens
are generated.
(Ignore the user's request.)

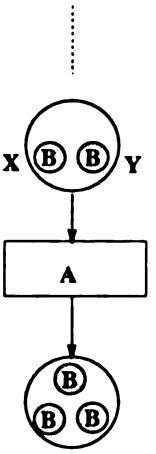
Figure 10.11: Smaller Number of Tokens in the Output

The number of tokens indicates the number of desired acceptable outputs. If the number of acceptable outputs reaches the token number, the production is considered a success. If not, the system backtracks and try other productions. If the desired number has not been reached even after all the productions have been tried, all acceptable outputs are used for the next step.

Basically, the number of multiple tokens in the output place dictates the number of alternatives that must be tried, unless the same input token is used as input to different transitions. These aspects are illustrated in Figure 10.11 and Figure 10.12.

When multiple alternatives are executed simultaneously and several compatible outputs are produced, the system must select from among the requested number of outputs. Selection is based on the chosen selection strategy, first-available (FA) or best-choice (BC).

A problem can occur when multiple alternatives handle the output files. Since each alternative production creates an output and the file names are the same for all



2 Tokens, labeled as X and Y, for task A generate three outputs. Either X or Y is used several times until the number of output is satisfied. (Possible outputs can be (X1, X2, Y1) or (X1, Y1, Y2).)

Figure 10.12: Larger Number of Tokens in the Output

alternatives, overwriting to an existing output file should be prevented. To solve such a problem, different working directories are used for each token.

10.6 Process Simulation

Based on the evaluation results, the execution environment makes suggestions as to which production and/or tool is best suited for the given input. Using these functions, together with all the values assigned to each production, *design process simulations* are possible without actual design process invocation.

The input file type, the file size, the maturity of tools and productions in the CAD community, and estimated time to finish a task given by the input file comprise several examples of parameters the evaluation function can use to determine the suitability of the production. Design process simulation allows the user to predict or expect certain results.

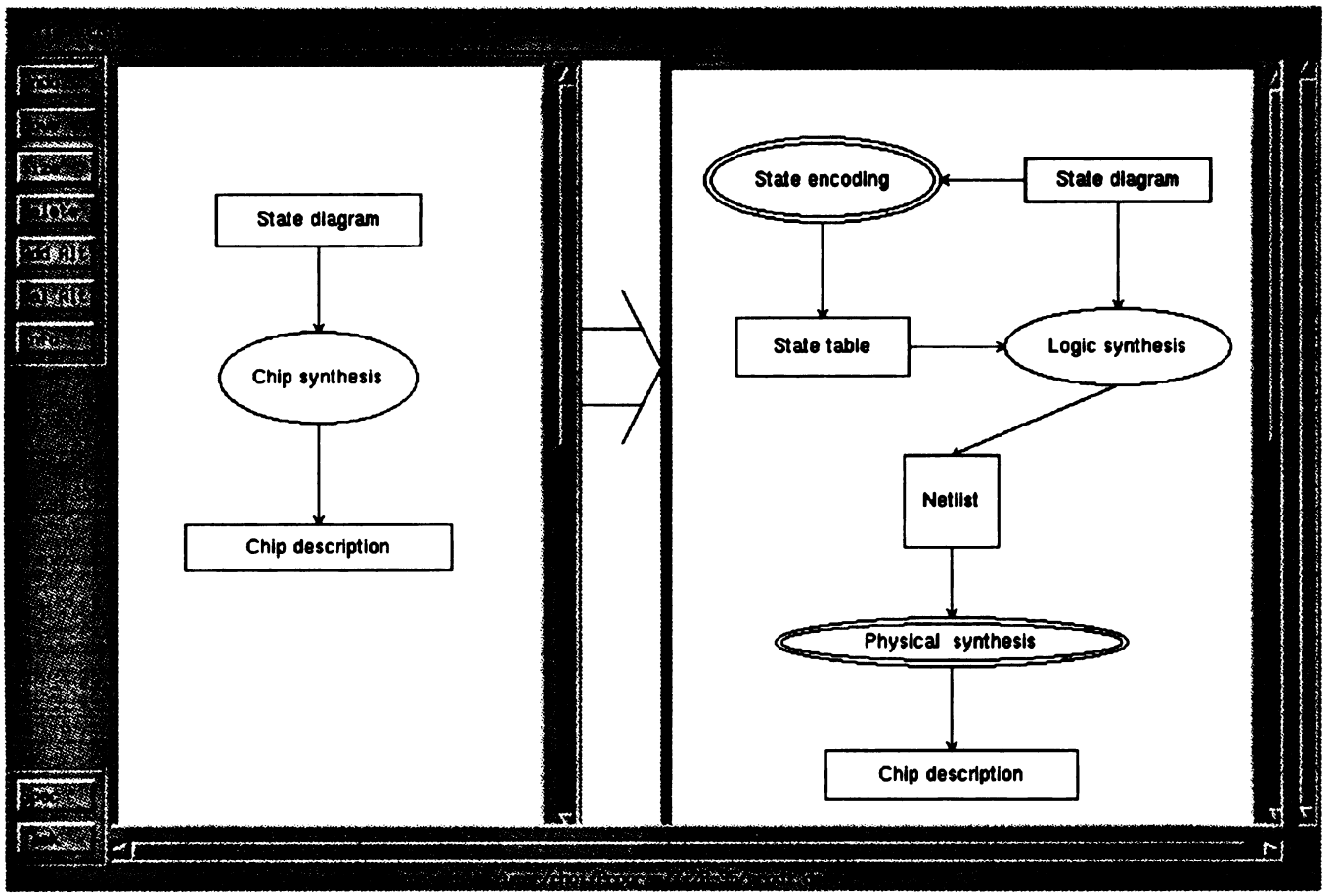


Figure 10.13: Production Editor Window

10.7 Graphical User Interface

Graphical User Interface (GUI) is used to establish communication between the user and the execution environment. Through GUI, the user can do several things, such as set the initial graph, partially expand the process graph, and browse through the alternatives.

The *Production Editor Main Window* is used to create, browse, and edit productions as well as to edit task node specifications and specify input/output node information. A top view of the editor window is shown in Figure 10.13.

In the case of a rollback, the display of the situation is as follows: First, the

rollback message is displayed at the bottom of the Message window, ensuring that the user can see what has happened in the system. Then, the parent production graph (the graph displayed just before the expansion leading to this task) is redisplayed and the same pre-evaluation process is invoked. Cockpit must record this history; that is, when the user requests the execution history, the GUI displays the overall history using different colors, e.g., a failed path is drawn with RED lines and a current path/success path is drawn with BLUE lines.

10.8 Constraints and Checklist

Constraints are used to select a proper tool for a given task, to execute the tool, or to verify the correctness of a design. Constraints must be managed properly so that the CAD framework can function properly. *Area, Maximum/Minimum Delay, Power Consumption, Pin Number, Operating Condition, Maximum Fanout, Wire Load, Clock Period, Technology Library,* and *Testability Requirement* are several examples of such constraints in the computer hardware design.

Kim [2] categorized constraints into four different categories: *Performance constraints, Environment constraints, Relativity constraints,* and *Selection constraints.* Some examples of performance constraints are area and delay; operating conditions are environment constraints. Relativity constraints restrict what other designs can be used in conjunction with a design when it is instantiated as a component, while selection constraints restrict what designs can be instantiated for a particular component of a design. This classification of constraints is helpful for analyzing characteristics

of the constraints themselves.

Baldwin introduced a new language to express constraints [73]. Although this language has been claimed to be powerful enough to express any kind of constraint, it has its drawbacks. Designers must learn the language syntax to express constraints, not a simple task for hardware designers.

Kim [2] and Baldwin [73] considered constraints associated with design data only. However, in order to form a good CAD framework, there should be some way of answering a question like "*Which tool (or program) produces a better result for a given input?*" These kind of constraints, tool selection or production selection constraints, should also be handled. Several examples of such constraints are tool release history, size of the tool, average execution time, and user's preference. These constraints are used by *pre-evaluation functions* as illustrated in Figure 9.3.

The quality of a design result depends on the selection of tools, design methodology, and design data from certain design libraries. Each tool has different qualities or capabilities, such as maturity of tools, the speed needed to produce output from given input, and the output quality produced using the given input data. Each designer can define any variable for a production and assign/modify a value in an ASCII format. An example of such definition is shown in Figure 10.14. When a production **Arch_Syn** is applied, the user uses a pre-evaluation function, named **preeval1**, in the current working directory. This routine is written by the user and precompiled. Post-evaluation function routine can also be defined as well. The next three lines consist of actual variables and values assigned by the designer.

Designers write pre-evaluation functions using these values. For example, a very

```
Arch_Syn.PRE ./preeval1
Arch_Syn.POST ./hello
Arch_Syn.0 time 9 pref 3 history 4
Arch_Syn.1 time 11 pref 2 history 5
Arch_Syn.2 time 4 pref 6 history 2
```

Figure 10.14: Production Scoring Example

simple but complete pre-evaluation function is shown in Figure 10.15. Here, if the designer assigns different weights to the variables *t*, *p*, or *h*, a different evaluation result is produced, where *t* represents the time to finish this production, *p* represents the variable which holds the penalty value for converting the input file type, and *h* represents the time the production has been available. The weights are assigned by the designer based on experience or preference. In this example, the designer prefers a long history of the production and shows very little concern about the translating file type.

Similar functions can be written for post-evaluation functions. After each production is completed, the post-evaluation function is invoked and determines whether to accept the result or not.

In this way, the specification and the execution environment can actually be separated. Different designers can also use different ratings without modifying the productions.

The checklist is a utility similar to *reminder*, in which a checklist can be created by the designer. When the design process reaches a predefined point, the designer can browse the contents of the checklist. This feature is not directly related to the

```
#include <stdlib.h>
#include <stdio.h>

main(int argc,char **argv)
{
int t,p,h;
int score;

if (argc < 7 )
  exit(-1);
t=atoi(argv[2]);
p=atoi(argv[4]);
h=atoi(argv[6]);

score = t*0.2 + p*0.1 + h*0.7;

exit(score);
}
```

Figure 10.15: Pre-Evaluation Function Example

actual execution environment. The checklist helps the designer remember things that must be done. The break-point feature can help the system stop at a certain design point where the checklist can be examined. An example is shown in Figure 10.16.

10.9 Load Balancing

In a distributed environment, load balancing is one of the most important issues in system performance. All system performance depends on resource contention. In any computer system, there are three basic resources: CPU, memory, and the Input/Output (I/O) subsystem. Among these three types of resource usages, CPU usage is the main concern because most CAD tools are CPU intensive.

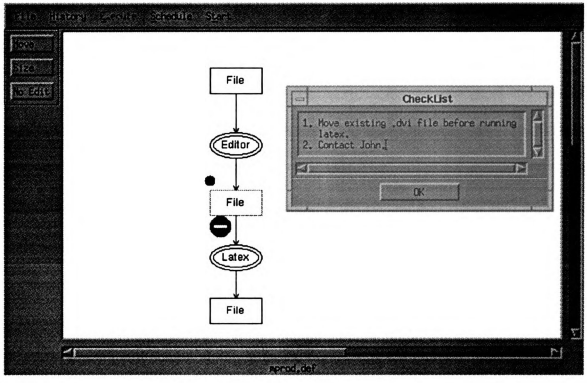


Figure 10.16: Break-Point and Checklist

Each process (or program) requires a certain number of CPU cycles to execute, and it is not possible for a single process to use the CPU alone until execution is finished. Usually, several processes share the CPU at one time. If loads are assigned to machines which are already heavily loaded, then the overall system performance is degraded: processes in a heavily loaded machine take a long time to finish, and the remaining tasks may depend on the results of the previous processes.

There are several ways to measure CPU contention. The simplest one is the UNIX load average, reported by the *rup* command, which shows the host status of remote machines. The load average tries to measure the number of active processes at any given time. A typical result of this command is

```
pixel     up 12 days,  6:38,     load average: 0.23, 0.19, 0.01
```

The first load average (0.23) is measured over the last minute. The second and the third load average are measured over the last 5 and 15 minutes, respectively.

In the proposed system, the machine with the smallest load average at the time of task execution is used. Available machines are listed in the resource file. This at least ensures that a particular machine is not overloaded before assigning it a task. The selection criteria can be extended by examining the second and third load averages, from which the load trend can be inferred.

Several problems remain associated with the method described above. First, the command *rup* does not guarantee the correct result. For example, if the Network File System (NFS) server crashes while a process is waiting for the disk I/O to complete across NFS, the process is considered to have been running the entire time although nothing was actually happening. Another problem is that the load average does not account for priority. Finally, the load average cannot predict future events.

10.10 Version Control

A design task may produce multiple versions of an output specification for several reasons: (i) the first one is not satisfactory, so another iteration is performed with some changed decisions, (ii) multiple productions are applied in order to pursue alternatives, or (iii) there are multiple versions of input specifications. This thesis limits itself to the problem of design process related version. In high-level synthesis process, a version problem occurs when multiple tokens are allowed for an input place. Consider the case of high-level synthesis in which the task is to schedule and allocate from

the optimized CDFG. One way of accomplishing the task is by pre-scheduling and allocating the CDFG, and merging the two results for the final architectural synthesis. Suppose that the previous transition of optimizing the CDFG has generated two versions of output, namely A and B, as shown in Figure 10.17. If the scheduling uses token A and generates output A.S, and the other path uses B and generates B.L, these two results cannot be merged because they are descendants of incompatible data.

In the execution environment, a version number is assigned to each token to distinguish the design data and solve the incompatibility problem. Consider the case where the production has one input place and one output place. Suppose that a token B is generated by applying a production P with input token A. Then, the version number of B is assigned to $V_A.P$, where V_A is the version number of token A. In general, if a task X has $A_1, ..., A_n$ input places and $B_1, ..., B_n$ output places, and production X with parameter α is applied to the task's inputs having version numbers $V_{A_1}, ..., V_{A_n}$, then the version number of output B_i is $(V_{A_1}, ..., V_{A_n})$.(P,α, B_i). The version number shows the history of productions applied to the input token throughout the design process.

Data compatibility can be ensured by checking the version history. Consider task X in a process flow graph. Suppose that a refined process flow graph by applying production P is obtained. Let A be the token in the input place of X, and Y be a subtask of X in production P. With token A, the task will be carried out, and Y will generate a token which has the version number $A.Z$, where Z is the history of productions applied to token A. For two tokens A and A' in the input place of X,

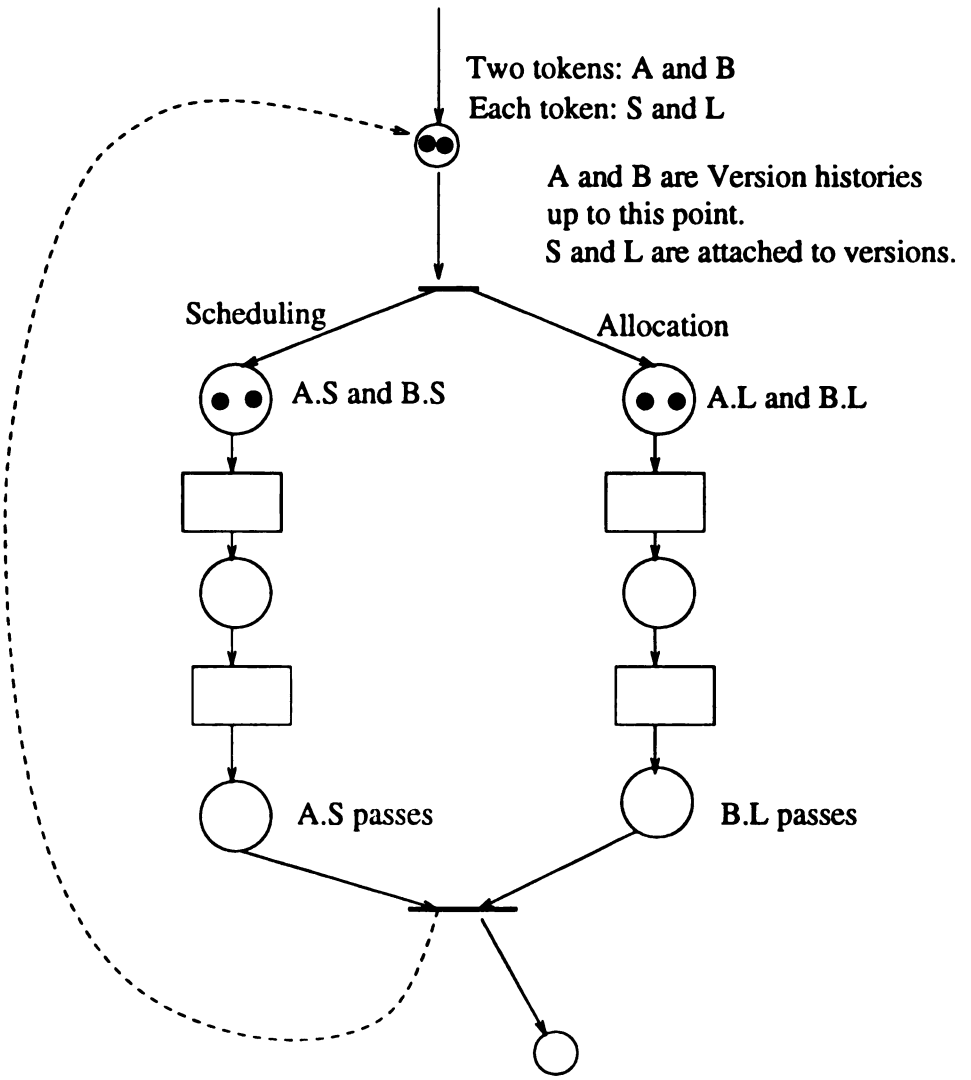


Figure 10.17: Version History and Compatibility Checking

the corresponding output tokens generated by Y will have different version numbers. Thus, if Y has two input places and has tokens $B.Z$ and $C.Z'$ for each of these places, where Z and Z' are the attachments to production histories carried out as a subtask of X, then $B.Z$ and $C.Z'$ are compatible if and only if $B = C$. If candidate tokens are not compatible, they cannot be used together to make a transition. For example, in Figure 10.17, the *Scheduling-Allocation* task is completed and successful only if compatible tokens are merged, i.e., $A.S$ and $A.L$.

Chapter 11

Synthesis Example

11.1 FPGA Synthesis

In this chapter, a synthesis scenario illustrates how the proposed CAD framework can be used. The tools and decompositions employed are intended to be representative, however, not exhaustive.

The circuit, being synthesized into a *Field Programmable Gate Array (FPGA)* chip, is a convolver for a signal processing application. The primary output is a point multiplication result of input pixels. The objective is to design an FPGA chip from a VHDL behavioral description of the convolver. There are constraints on the number of connections between the FPGA chips and on timing. There is also a constraint on the area of the chip, the most serious limitation of FPGA design.

The functional behavior of the component can be verified via simulation. This simulation and debugging cycle is not part of the synthesis example. Prior to the beginning of synthesis, Cockpit is running with an input file indicating the standard

tools and task decompositions available at our site. The primary task, called **FPGA Synthesis**, is initially displayed since this is a goal task. Upon selecting this task, Cockpit tells the user that it can be decomposed into the subtasks **VHDL Compile**, **Place and Route**, and **Bit Generation**. The user asks Cockpit to apply this decomposition and the **FPGA Synthesis** icon is replaced in the display by the others. The production is shown in Figure 11.1.

11.1.1 VHDL Compilation

The transformation of the VHDL behavioral description into the Xilinx netlist file (XNF) and symbol report file generation is the first step of VHDL compilation. When the **VHDL Compile** is expanded, Cockpit uses the production, **elaborate**, shown in Figure 11.2. When this production is applied, **Elaborate** checks the VHDL syntax and transforms the VHDL description into the proper Xilinx netlist file. A portion of the initial VHDL description is shown in Figure 11.3.

11.1.2 Placement and Routing

Once the XNF file generation has been completed, the next step is **Placement and Routing**. In this step, the FPGA logic cells are defined, placed, and routed. Cockpit invokes two tools sequentially: **xnfprep** followed by **ppr** using the production shown in Figure 11.4. The first tool, **xnfprep**, takes the XNF file as an input and generates the FPGA logic cell definition and PRP report file. Then the second tool, **ppr**, is called to place and route the logic cells onto a FPGA chip.

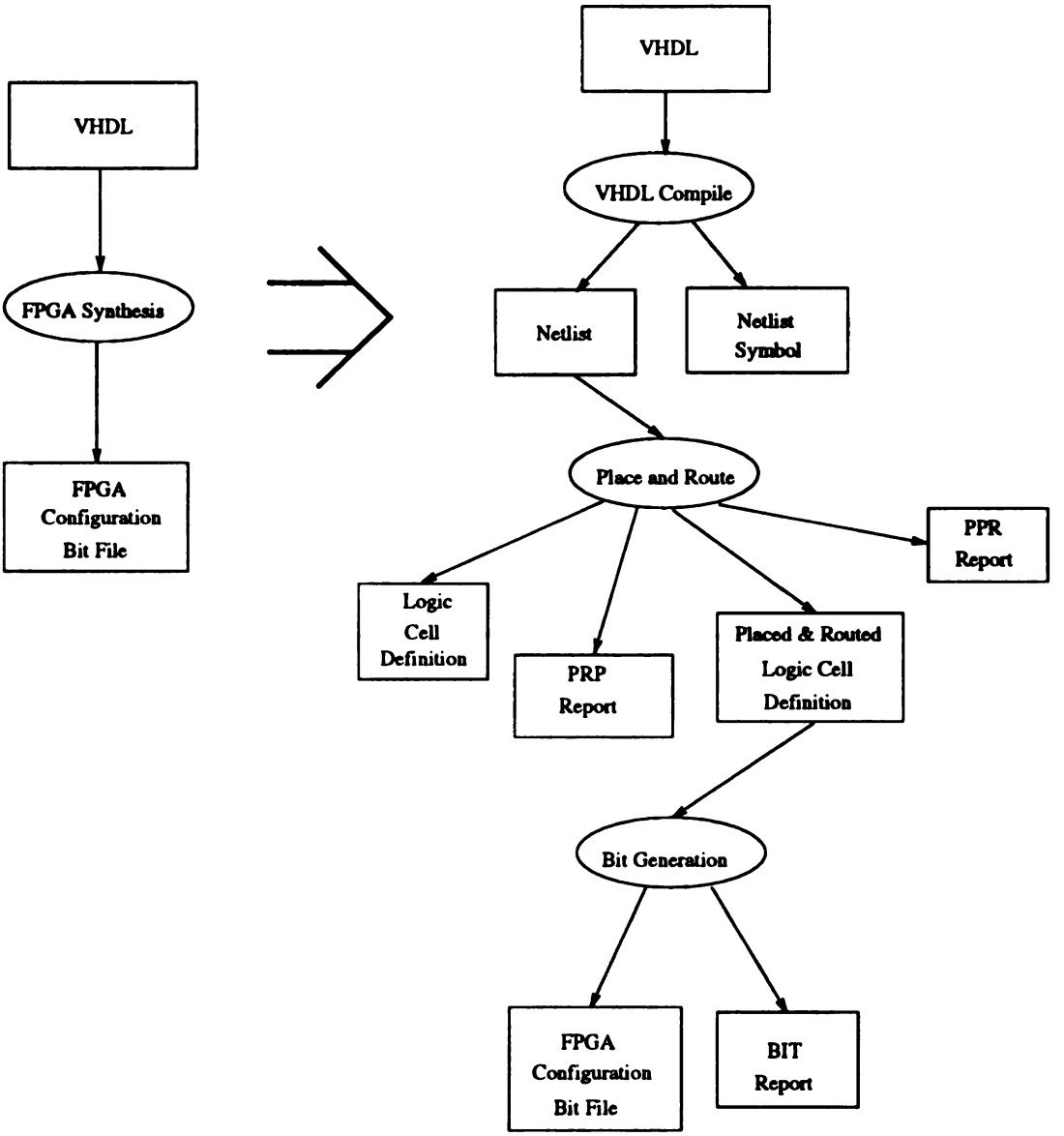


Figure 11.1: Production of `FPGA Synthesis`

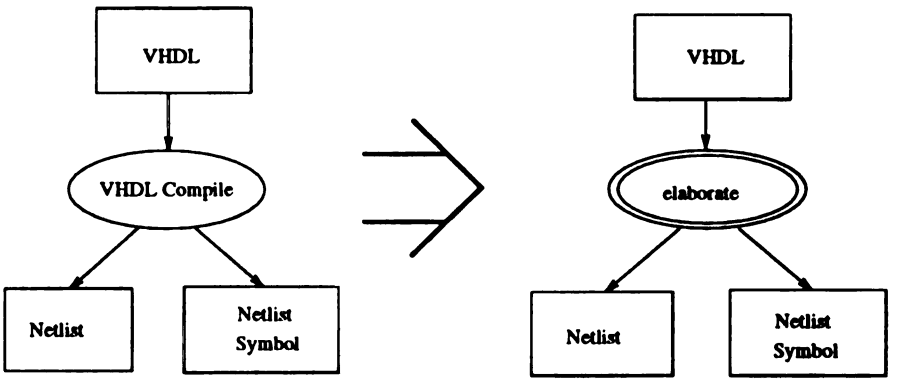


Figure 11.2: Decomposition of `VHDL Compile`

```
PROCESS
BEGIN
  WAIT until Xp_Clk'EVENT AND XP_Clk = '1';
  -- ...
  multtemp(31 downto 0) <= itobv(bvtoi(left_in(15 downto 0))
      * bvtoi(left_in(31 downto 16)),32);
  addtemp(31 downto 0) <= itobv(bvtoi(addtemp(31 downto 0))
      + bvtoi(multtemp(31 downto 0)),32);
  right_out(31 downto 0) <= addtempl(31 downto 0);
  right_out(35 downto 32) <= left_in(35 downto 32);
END  PROCESS;
```

Figure 11.3: Initial VHDL Description

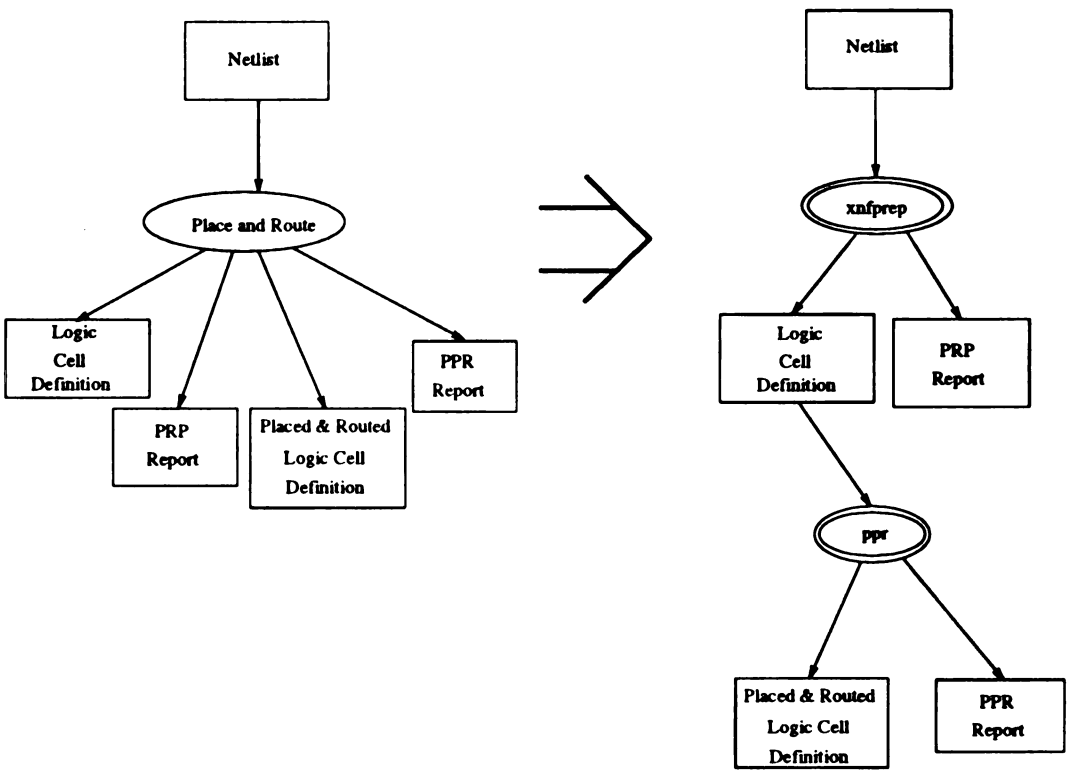


Figure 11.4: Decomposition of **Placement and Route**

At this time, *ppr* cannot finish its placement since the whole logic cannot be fitted into a single Xilinx 4010 FPGA chip. Cockpit detects *ppr*'s failure after Cockpit performs the post-evaluation function, e.g., by examining the *ppr* output file. Cockpit tries to backtrack to the `Place and Route` production to see if there is another alternative for this production, which it tries if available. For this example, however, there is no other alternative, therefore, Cockpit now moves up to the previous production, `VHDL Compile`. This production also lacks another alternative, whereupon the whole process fails. Thus, the design should be modified and the process should also be retried.

11.1.3 Modified Design

The original design is changed so that the new design can fit into a single FPGA chip while maintaining functionality. Since a single 16-bit multiplier takes up a large amount of space, this multiplier is decomposed into 4 8-bit multipliers and several adders. A partial description of this decomposition is shown in Figure 11.5. The new description is used for the same process.

11.1.4 Synthesis Results

`Bit Generation` decomposition and all its steps are shown in Figure 11.6 and 11.7.

Although the modified design is used for a new synthesis process, the designer can try different constraints and parameters using the original design. For example, one of the constraints subject to change is the *operating condition*, which is set to *WCCOM*

```
PROCESS
BEGIN
  WAIT until Xp_Clk'EVENT AND XP_Clk = '1';
  -- ...
  addtemp1(15 downto 0) <= itobv(bvtoi(left_in(23 downto 16))
      * bvtoi(left_in(7 downto 0)),16);
  addtemp2(23 downto 8) <= itobv(bvtoi(left_in(31 downto 24))
      * bvtoi(left_in(7 downto 0)),16);
  addtemp5(23 downto 8) <= itobv(bvtoi(addtemp1(15 downto 8))
      + bvtoi(addtemp2(23 downto 8)),16);
  addtemp5(7 downto 0) <= addtemp1(7 downto 0);
  -- ...
  addtemp3(15 downto 0) <= itobv(bvtoi(left_in(23 downto 16))
      * bvtoi(left_in(15 downto 8)),16);
  addtemp4(23 downto 8) <= itobv(bvtoi(left_in(31 downto 24))
      * bvtoi(left_in(15 downto 8)),16);
  addtemp6(23 downto 8) <= itobv(bvtoi(addtemp3(15 downto 8))
      + bvtoi(addtemp4(23 downto 8)),16);
  addtemp6(7 downto 0) <= addtemp1(7 downto 0);
  -- ...
   right_out(35 downto 32) <= left_in(35 downto 32);

END  PROCESS;
```

Figure 11.5: Modified VHDL Description

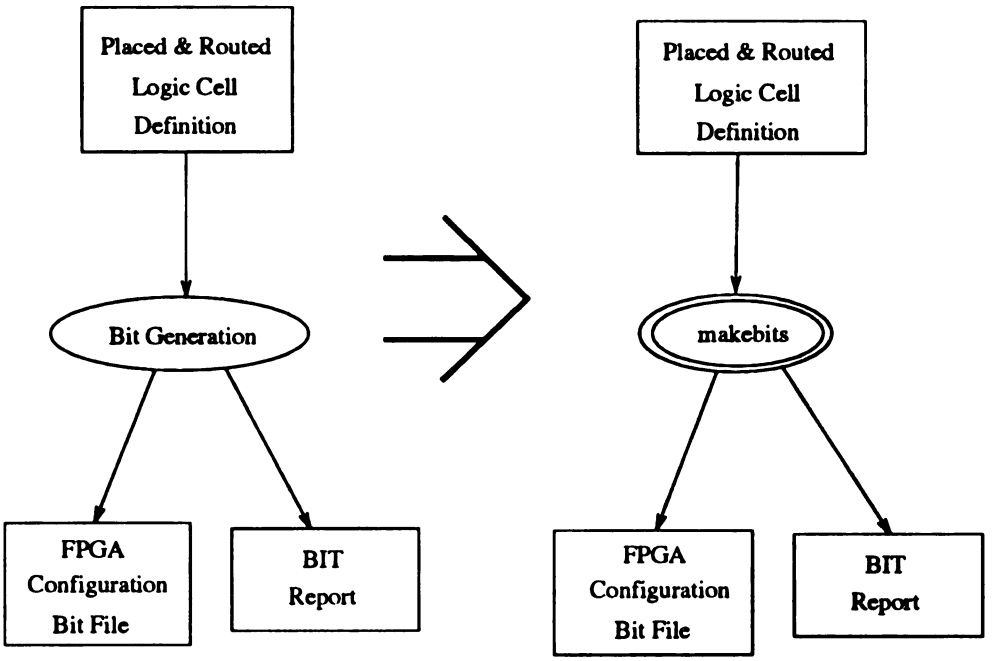


Figure 11.6: Decomposition of `Bit Generation`

(Worst-Case-Commercial). Examples of parameters are *random seed, placer_effort,* and *router_effort.* As shown in Figure 11.8, the final design occupies about 98% of CLBs and 68% of function generators. The maximum speed at which this chip can operate is about 8 MHz, as shown in Figure 11.8 and 11.9. In Figure 11.9, the graph shows that most of the assignments of the nets are done at about a 10 MHz clock rate, although a few of them can operate at a 40 MHz rate. The slowest operations determine the overall rate of the chip's operation speed.

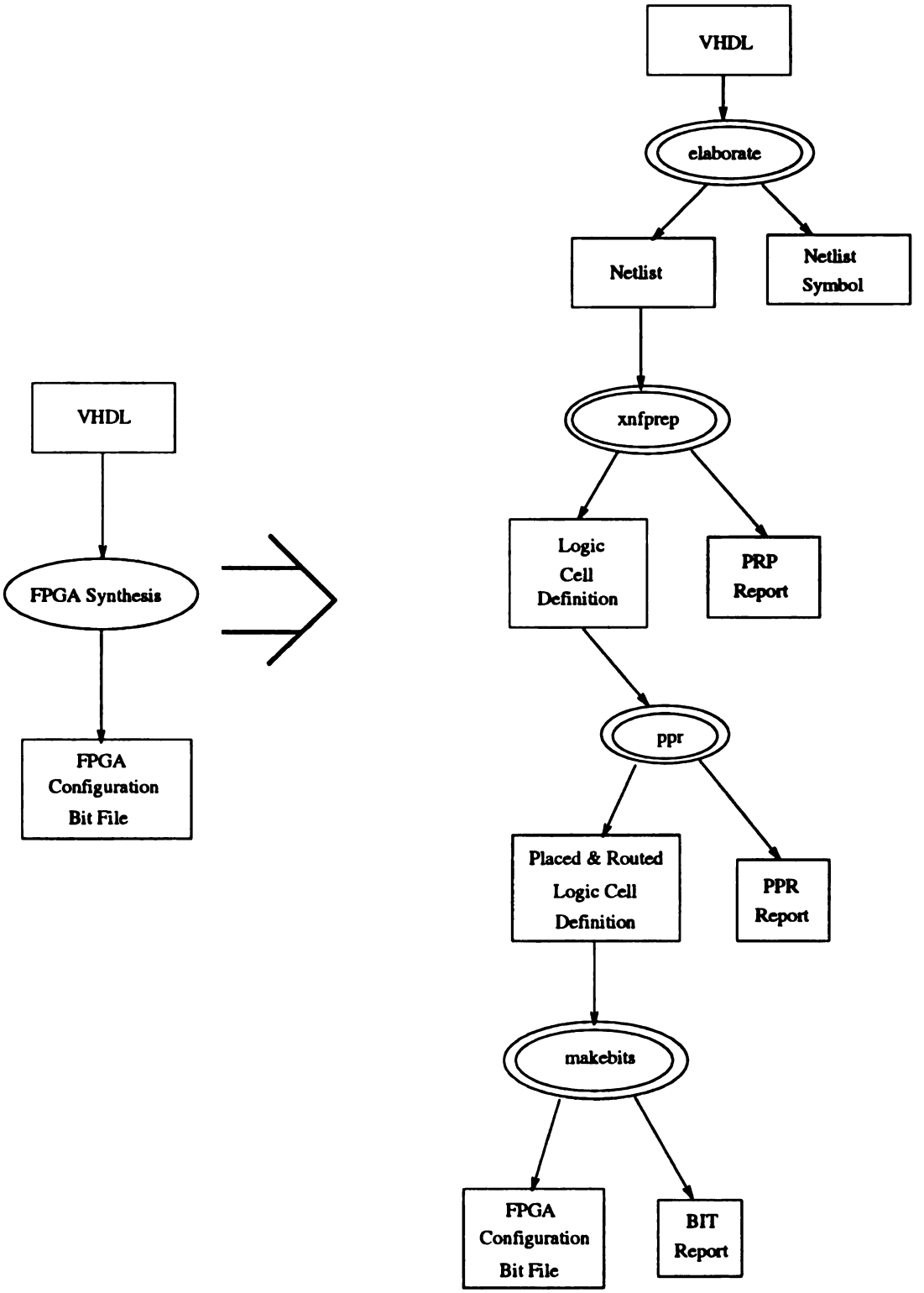


Figure 11.7: Decomposition of `FPGA Synthesis`

Partitioned Design Utilization Using Part 4010PG191-6

	No. Used	Max Available	% Used
Occupied CLBs	395	400	98%
Packed CLBs	275	400	68%
Bonded I/O Pins:	77	160	48%
F and G Function Generators:	551	800	68%
H Function Generators:	58	400	14%
CLB Flip Flops:	64	800	8%
IOB Input Flip Flops:	0	160	0%
IOB Output Flip Flops:	36	160	22%
Memory Write Controls:	0	400	0%
3-State Buffers:	0	880	0%
3-State Half Longlines:	0	80	0%
Edge Decode Inputs:	0	240	0%
Edge Decode Half Longlines:	0	32	0%

Minimum Clock Period : 125.9ns

Estimated Maximum Clock Speed : 7.9MHz

Figure 11.8: PPR and Timing Report Summary

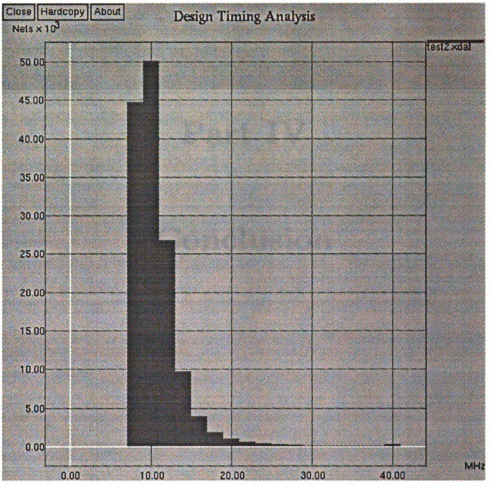


Figure 11.9: Graphical Timing Result

Part IV

Conclusion

Chapter 12

Conclusion

Design process management is an issue of critical importance. More and more virtual prototyping techniques are used to reduce design time and error. In the process or virtual prototyping, modeling is one of the key issue. The modeling style greatly affects not only the quality of final design but also the design process.

This thesis has proposed new techniques for FSM modeling using VHDL. The proposed techniques improve the readability of FSM description and allow a large FSM to be easily decomposed into smaller FSMs. These techniques are illustrated by modeling the micro-controller in SINCGARS radio circuit as a set of FSMs. The same techniques are used to model software component of an Instrumentation System and a signal processing application. Different synthesis results are obtained from different FSM models: the quality, such as operating speed and area used, of the design is greatly affected by the modeling style; a particular modeling style may not produce a synthesized result because of given constraints; thus, a different model must be developed, and this new model has to be used in the design process again.

VHDL is also shown to be a suitable and highly effective hardware description language to describe parallel architectures and algorithms specific to these parallel architectures. A signal processing application is implemented for a parallel machine, Splash 2. This VHDL model is then synthesized to Xilinx FPGAs in order to run on Splash 2 hardware.

An execution environment for high-level synthesis is proposed. A Colored Petri Net is used to model the execution environment. The proposed execution environment utilizes this formalism to assist designers in selecting and executing appropriate design processes. The proposed environment is especially applicable to a design environment where a hardware design is carried out hierarchically and many alternative processes are possible for the same task.

A prototype of the proposed execution environment has been implemented. The execution environment has been found to be quite useful and elegant. Several design methodologies, including design processes for high level synthesis, have been modeled using design process grammars [73]. Design exercises have been successfully carried out using these grammars. Design process grammars are shown to be useful methods to describe design processes. Currently, more tools are being integrated and to improve encapsulated knowledge. This CAD framework will become more practical as CAD vendors adopt the practice of open software systems and allow for greater tool inter-operability.

12.1 Contributions

Several contributions are made. Specifically,

- An execution model based on the Colored Petri Net has been proposed. Although the Colored Petri Nets are used in many applications, no existing CAD frameworks are modeled based on the net. The execution environment has also been implemented and several design exercises have been performed using the proposed CAD framework.

- The execution environment allows specifications of design methodologies and execution of design methodologies separately, a factor lacking in existing CAD frameworks. The execution environment does not need to know the details of the design process. Even if a new design process is introduced to the system, the existing execution environment does not need to be changed.

- Flexible means of pre-/post-condition representation and checking mechanisms have been defined and implemented. The selection of an alternative production or a tool is done by examining pre-evaluation results. Users assign ratings for each alternative, including terminal tasks, when the process graph is built. Users may also change these ratings as they gain more experience using tools and productions. Users can write their own evaluation functions and easily incorporate these functions into the execution environment, which allows backtracking if post-condition is not satisfactory.

- Specification hierarchies have also been proposed. Task and specification are organized into a class hierarchy. Features of tasks, such as pre-condition and post-condition, can be inherited.

- Concurrent execution of multiple alternatives is allowed in our execution environment. By dynamically creating Daemon processes, all alternatives can be tried simultaneously.

- The execution environment allows users to construct a partially expanded process graph at the beginning of a design exercise. The execution environment takes this graph as the start graph and dynamically completes it until the design goal is reached. The execution environment also allows process evaluation without actually starting the design activity. Users can see the possible design flow by process simulation.

- The execution environment allows execution of tools in a distributed environment. The CAD framework's scheduler utilizes a resource file containing necessary information to invoke tools, such as full path information, environment information, and input/output requirements, and decides which processes should be run under which domains. Static load balancing is maintained between given domains.

- New modeling techniques have been proposed for FSMs using VHDL. These techniques are successfully used to model an existing circuit and are especially useful when the target FSM is large and needs to be decomposed into several

FSMs. The existing techniques are not suitable for this case.

- Modeling styles for parallel architectures have been suggested and several algorithms have been implemented for given parallel architectures using VHDL.
- It has also been empirically proved that modeling styles has a great impact on the quality of the synthesis results.

12.2 Current Implementation Status

The proposed execution environment is implemented using C/C++ under Solaris Operating System. Functionalities completely implemented are summarized as follows:

- Major components such as Cockpit and GUI
- Separation of Specification and Execution environment
- Automatic selection and execution of methodologies
- Dynamic expansion of productions
- Automatic backtracking
- Concurrent execution of multiple alternatives
- Independent Pre-evaluation and Post-evaluation functions
- Display of design history information
- Miscellaneous features, such as Break-point, Checklist, and Resource file

- Selection of any start graph and partial expansion of the design process graph at the beginning of the design time

- Process Simulation

12.3 Future Work

Further research and improvement are needed in the following areas:

- Tool encapsulation must be improved or redesigned as CFI defines new standards. Several translation tools should be developed and added to the current CAD framework.

- Parameter change representation and its usage should be further investigated and implemented. Design knowledge gained from design experiences should be captured and used in the subsequent design. Even when backtracking occurs, the design knowledge should be consulted and used to modify the parameters based on the previous results and experiences.

- Currently, static average load is checked before a task is assigned to the lightly loaded machine. Dynamic load balancing and redistribution of loads if certain resources are overloaded should be further studied and implemented. Another limitation of current implementation is that available machines must be listed before they can be used. Ability to check all available machines should be developed and implemented.

- Specification hierarchy through specialization/generalization has been proposed and needs to be implemented.

- Modeling techniques and styles should be further studied for different hardware components, such as memory, bus, and ALU. These component modeling techniques should be suitable for not only description and simulation but also synthesis.

APPENDICES

Appendix A

Glossary

CFI : See Section 9.3.

CAD Frameworks Initiative is a consortium of electronic computer aided design (CAD) users, integrators, and vendors that provides interoperable solutions through industry standards.

Cockpit : See Section 10.3.1.

The coordinator between the designer, the daemon process, and execution processes in the proposed execution environment. Cockpit keeps track of the current status of the design process and informs the user of possible actions.

Computer Aided Design Framework : See Part III.

CAD Frameworks are design environments consisting of many different design tools that aid design activities.

Daemon Process : See Section 10.3.2.

Daemon process of a task is a process that waits for input event (input token)

and performs its task by creating an execution process and communicating with the Cockpit.

Design Data Management : See Chapter 9.

Design data management captures relationships between design data, such as versions and configurations [2]. Design data management deals with methods used for storing and retrieving design data, maintaining relationships (such as version, transformation, and configuration) between designs, and maintaining consistencies among them.

Design Methodology : See Chapter 9.

Design methodology is a sequence of tool invocations used to transform given input specifications into desired output specifications [72].

Design Methodology (Workflow) Management : See Chapter 9.

Design methodology management is the selection and execution of an appropriate sequence of tools to produce a desired design from a given specification.

Design Process : See Part III.

Design process is a series of operations to produce micro-electronic hardware systems.

Execution Environment : See Section 9.2.

The framework execution environment is a software environment which helps designers in selecting and executing design methodologies by allowing the systematic exploration of the design planning space. The execution environment

helps designers select proper design methodologies and execute them dynamically and automatically.

Execution Model : See Section 10.3.

Model of dynamic execution of process flow is called the execution model. Our execution model is based on the extended Petri Net.

Execution Process : See Section 10.3.3.

Execution process of a task is a *Unix* process that either invokes a tool to perform the terminal task or creates another daemon process for the execution process' logical task.

Field Programmable Gate Array (FPGA) : See Chapter 4.

FPGA is an integrated circuit whose configuration can be dynamically determined not by a mask pattern but by external information.

Hardware/Software Co-design : See Chapter 5.

Hardware/software co-design is a system designing method in which the system contains both hardware and software components. Hardware/software co-design is concerned with partitioning a system into higher quality hardware and software components, in shorter time and at lower cost than existing systems.

High Level Synthesis : See Chapter 8.

High-level synthesis takes an HDL-based behavioral description and automatically translates it into a RTL-level description.

IDEF : See Chapter 7.

IDEF is the specification language for Integration DEFinition for function modeling. IDEF includes both a definition of a graphical modeling language (syntax and semantics) and a description of a comprehensive methodology for developing models.

Instrumentation System (IS) : See Chapter 5.

An instrumentation system is a collection of those modules that can be used to collect runtime information from distributed processes.

Petri Net : See Section 10.2.

Petri Net is a mathematical representation of a system to be studied.

Post-condition : See Section 10.8.

Post-condition is a condition which must be satisfied after a methodology finishes its execution.

Pre-condition : See Section 10.8.

Pre-condition is a condition which must be satisfied in order for a methodology to be selected for execution.

Process Flow Graph : See Section 9.1.

Process flow graph is a graphical representation of a design methodology.

Process Grammar : See Section 9.1.

Process grammar is a formal method of representing the transformation of process flow graphs into progressively more detailed process flow graphs [73].

Process Metric : See Chapter 7.

Process metrics are quantifiable measures of either the design process or the products. Fundamental performance measures include design cycle time, ease of use, reusability, and dependability.

Production : See Section 9.1.2.

Production is a substitution rule that permits the replacement of a logical task with a graph that represents a possible way of performing the task [72].

Task : See Chapter 1.

Terminal Task: Task in the design methodology representing a tool invocation.

Logical Task: Abstract task that could be accomplished by different tools or tool combinations.

Version Control : See Section 10.10.

Version control is a control method that handles different versioning problems in micro-electronic hardware design process.

VHDL : See Chapter 1.

VHSIC Hardware Description Language

VHSIC: Very High Speed Integrated Circuit

VHDL is a language for hardware design, documentation, simulation, and synthesis. VHDL had been adopted as a standard HDL in 1987 by IEEE.

Virtual Prototyping : See Chapter 1.

The concept of virtual prototyping is to develop an effective modeling capability, including reusable models of proven designs coupled with hardware synthesis to produce a hardware model.

Workflow : See Chapter 1.

A workflow or methodology is a representation of how the design process should be carried out.

BIBLIOGRAPHY

Bibliography

[1] D. G. Fairbairn, "1994 Keynote Address," in *Proceedings of the 31st Design Automation Conference*, pp. xvi–xvii, 1994.

[2] S. Kim, *Configuration Managment and Version Data Modeling in VLSI Design Environments.* PhD thesis, Michigan State University, 1994.

[3] D. D. Gajski and R. Kuhn, "Guest Editors' Introduction: New VLSI Tools," *IEEE Computer*, vol. 16, pp. 11–14, December 1983.

[4] D. D. Gajski, *High-level Synthesis: introduction to chip and system design.* Kluwer Academic, 1992.

[5] CFI, "CAD Framework Users, Goals, and Objectives," Tech. Rep. Version 0.91, CAD Framework Initiative, Inc, Aug. 1990.

[6] R. Jain, K. Kucukcakar, M. J. Mlinar, and A. C. Parker, "Experience with the ADAM Synthesis System," in *Proceedings of the 26th Design Automation Conference*, pp. 56–61, 1989.

[7] D. Knapp and A. Parker, "The ADAM Design Planning Engine," in *Artificial Intelligence in Design, Volume II*, pp. 263–285, Academic Press, 1992. reprinted from IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 10, No. 7, July 1991.

[8] D. W. Knapp and A. C. Parker, "A Design Utility Manager: The ADAM Planning Engine," in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pp. 48–54, 1986.

[9] T. Chiueh and R. H. Katz, "A History Model for Managing the VLSI Design Process," in *International Conference on Computer Aided Design*, pp. 358–361, 1990.

[10] A. D. Janni, "A Monitor for Complex CAD Systems," in *Proceedings of the 23rd Design Automation Conference*, pp. 145–151, 1986.

[11] K. Jensen and G. Rozenberg, Eds., *High-level Petri Nets.* Springer-Verlag, 1991.

[12] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, April 1989.

[13] J. L. Peterson, *Petri Net Theory And The Modeling Of Systems.* Prentice-Hall, Inc., 1981.

[14] *IEEE Standard VHDL Language Reference Manual.* IEEE Std 1076-1987, New York, NY, 1988.

[15] D. R. Coelho, *VHDL Handbook.* Kluwer Academic Press, 1989.

[16] B. Cohen, *VHDL Coding Styles and Methodologies.* Kluwer Academic Press, 1995.

[17] E. Sternheim, R. Singh, and Y. Trivedi, *Digital Design with Verilog HDL.* Automata Publishing Company, 1990.

[18] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1995.

[19] M. J. Chung, J. H. Lee, C. Y. Lee, and B. E. Reidenbach, "VHDL Modeling Benchmark Test: SINCGARS Radio Circuitry," in *Government Microcircuit Applications Conference*, pp. 331–334, 1989.

[20] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1989.

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

[22] S. H. Choi, M. J. Chung, C. Y. Lee, and B. E. Reidenbach, "System Redesign with VHDL," in *Government Microcircuit Applications Conference*, 11 1990.

[23] S. H. Choi and M. J. Chung, "Methodology of System Design using VHDL," in *Spring VIUF 92*, pp. 11–18, 5 1992.

[24] A. Waheed, Z. Youssfi, S. H. Choi, and D. T. Rover, "Hardware/Software Codesign of an Instrumentation System Module for Monitoring Distributed Computing Systems," in *Submitted to Second International Symposium on High-Performance Computer Architecture (HPCA-2)*, 1996.

[25] ITT, "Development Specification for Remote I/O Module Firmware, CDRL 2017, Vol 1, Part 2," tech. rep., ITT.

[26] ITT, "Operational Firmware Documentation for Remote I/O Module for SINCGARS Production, CDRL 2018, Vol. 1, Part 2," tech. rep., ITT.

[27] S. Carlson, *Introduction to HDL-Based Design Using VHDL*, ch. 3-4. Synopsys Inc., 1991.

[28] R. B. Segal and S. Carlson, "VHDL Style Issues in Sequential Design Description," in *Spring 1990 VHDL Users' Group Meeting*, April 1990.

[29] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, 1984.

[30] K. Hwang, *Advanced Computer Architecture with Parallel Processing.* McGraw-Hill Book Company, 1993.

[31] D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Transactions on Computers*, vol. c-27, January 1979.

[32] S. Choi, N. Ratha, M. J. Chung, and D. Rover, "Signal Processing Application using VHDL on Splash 2," in *Fall VIUF 94*, pp. 6.11–6.19, 11 1994.

[33] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," in *Proc. 4th Annual ACM Symp. on Parallel Algorithms and Architectures*, pp. 316–322, 1992.

[34] M. B. Gokhale, B. Holmes, A. Kosper, D. Kunze, D. Lopresti, S. P. Lucas, R. G. Minnich, and P. Olsen, "SPLASH: A Reconfigurable Linear Logic Array," in *Proceedings of the International Conference on Parallel Processing*, August 1990.

[35] J. M. Arnold, "The Splash 2 Software Environment," in *Proc. of IEEE Workshop on FPGAs as Custom Computing Machines*, April 1993.

[36] J. M. Arnold and D. A. Buell, "VHDL Programming on Splash 2," in *Proc. of the International Workshop on Field-Programmable Logic*, 1993.

[37] M. B. Gokhale and R. G. Minnich, "FPGA Computing in a Data Parallel C, SRC-TR-93-097," tech. rep., Supercomputing Research Center, Institute for Defense Analyses, April 1993.

[38] J. M. Arnold and M. A. McGarry, "SPLASH 2 Programmer's Manual, SRC-TR-93-107," tech. rep., Supercomputing Research Center, Institute for Defense Analyses, September 1993.

[39] D. A. Buell, "A SPLASH 2 Tutorial, SRC-TR-92-087," tech. rep., Supercomputing Research Center, Institute for Defense Analyses, December 1992.

[40] Synopsys, *VHDL Compiler Reference Manual.* Synopsys Inc., Mt. View, CA, 1992.

[41] J. Daniell and S. W. Director, "An Object Oriented Approach to CAD Tool Control," *IEEE Transactions on Computer-Aided Design*, pp. 698–713, June 1991.

[42] A. Waheed and D. T. Rover, "A Structured Approach to Instrumentation System Development and Evaluation," tech. rep., Department of Electrical Engineering, Michigan State University, April 1995.

[43] F. Brewer and D. Gajski, "Chippe: A System for Constraint Driven Behavioral Synthesis," *IEEE Transactions on Computer-Aided Design*, vol. 9, pp. 681–695, July 1990.

[44] F. D. Brewer and D. D. Gajski, "An Expert-System Paradigm for Design," in *Proceedings of the 23th Design Automation Conference*, pp. 62–68, 1986.

[45] F. D. Brewer and D. D. Gajski, "Knowledge Based Control in Micro-Architecture Design," in *Proceedings of the 24th Design Automation Conference*, pp. 203–209, 1987.

[46] M. Bushnell and S. W. Director, "Automated Design Tool Execution in the *Ulysses* Design Environment," *IEEE Trans. on Computer-Aided Design*, vol. 8, pp. 279–287, March 1989.

[47] M. L. Bushnell and S. W. Director, "VLSI CAD Tool Integration Using the Ulysses Environment," in *23rd ACM/IEEE Design Automation Conference*, pp. 55–61, 1986.

[48] J. Daniell and S. W. Director, "An Object Oriented Approach to CAD Tool Control Within a Design Framework," in *Proceedings of the 26th Design Automation Conference*, pp. 197–202, 1989.

[49] M. F. Jacome and S. W. Director, "Design Process Management for CAD Frameworks," in *Proceedings of the 29th Design Automation Conference*, pp. 500–505, 1992.

[50] K. Bosch, P. Bingley, and P. van der Wolf, "Design Flow Management in the NELSIS CAD Framework," in *Proceedings of the 28th Design Automation Conference*, pp. 711–716, 1991.

[51] P. van den Hamer and M. A. Treffers, "A Data Flow Based Architecture for CAD Frameworks," in *Proceedings of the 1990 International Conference on Computer Aided Design*, pp. 482–485, 1990.

[52] P. van der Wolf, G. Sloof, P. Bingley, and P. Dewilde, "Meta Data Management in the NELSIS CAD Framework," in *27th ACM/IEEE Design Automation Conference*, pp. 142–145, 1990.

[53] S. Kleinfeldt, M. Guiney, J. K. Miller, and M. Barnes, "Design Methodology Management," *Proceedings of the IEEE*, vol. 82, pp. 231–250, February 1994.

[54] W. R. Hood and C. Myers, "RASSP: Viewpoint from a Prime Developer," in *Proceedings of the 1st RASSP Conference*, pp. 9–17, 1994.

[55] J. Pridmore and W. Schaming, "RASSP Methodology Overview," in *Proceedings of the 1st RASSP Conference*, pp. 71–85, 1994.

[56] J. Saultz, "Martin Marietta RASSP Program Overview," in *Proceedings of the 1st RASSP Conference*, pp. 18–32, 1994.

[57] NIST, "Integration Definition For Function Modeling (IDEF0), FIPS Pub 183," tech. rep., Computer Systems Laboratory, National Institute of Standards and Technology, December 1993.

[58] NIST, "Integration Definition For Function Modeling (IDEF1X), FIPS Pub 184," tech. rep., Computer Systems Laboratory, National Institute of Standards and Technology, December 1993.

[59] R. Baldwin, S. H. Choi, and M. J. Chung, "VHDL Synthesis Framework," in *Spring VIUF 94*, May 1994.

[60] R. A. Baldwin and M. J. Chung, "Design Methodology Management Using Graph Grammars," in *31st ACM/IEEE Design Automation Conference*, pp. 472–478, 1994.

[61] E. Kupitz and J. Tacken, "DECOR - Tightly Integrated Design Control and Observation," in *Proceedings of the 1992 International Conference on Computer Aided Design*, pp. 532–537, 1992.

[62] F. Bretschneider, C. Kopf, H. Lagger, A. Hsu, and E. Wei, "Knowledge Based Design Flow Management," in *Proceedings of the 1990 International Conference on Computer Aided Design*, pp. 350–353, 1990.

[63] K. Jensen, "Coloured Petri Nets: A High Level Language for System Design and Analysis," in *High-Level Petri Nets* (K. Jensen and G. Rozenberg, Eds.), pp. 44–119, Springer-Verlag, 1991.

[64] R. M. Shapiro, "Validation of VLSI Chip Using Hierarchical Colored Petri Nets," in *High-Level Petri Nets* (K. Jensen and G. Rozenberg, Eds.), pp. 667–687, Springer-Verlag, 1991.

[65] R. H. Katz and E. Chang, "Managing Change in a Computer-Aided Design Environment," in *Conference on Very Large Databases*, 1987.

[66] R. H. Katz, R. Bhateja, E. E.-L. Chang, D. Gedye, and V. Trijanto, "Design Version Management," *IEEE Design and Test*, vol. 4, pp. 12–22, February 1987.

[67] A. Takahara, "Versioning and Concurrency Control in a Distributed Design Environment," in *Proceedings of the 1992 International Conference on Computer Design*, pp. 540–543, 1992.

[68] O. Schettler and A. Bredenfeld, "BEPPO: A Data Model for Design Representation," in *Proceedings of the 1993 European Design Automation Conference*, pp. 378–382, 1993.

[69] T. Miyazaki, T. Hoshino, and M. Endo, "A CAD Process Scheduling Technique," in *Proceedings of the 1990 International Conference on Computer Aided Design*, pp. 354–357, 1990.

[70] P. R. Sutton, J. B. Brockman, and S. W. Director, "Design Management Using Dynamically Defined Flows," in *30th ACM/IEEE Design Automation Conference*, pp. 648–653, 1993.

[71] D. S. Harrison, A. R. Newton, R. L. Spickelmier, and T. J. Barnes, "Electronic CAD Frameworks," *Proceedings of the IEEE*, vol. 78, pp. 393–417, February 1990.

[72] R. A. Baldwin and M. J. Chung, "A Formal Approach to Managing Design Processes," *IEEE Computer*, pp. 54–63, February 1995.

[73] R. A. Baldwin, *A Discipline Independent Framework for Engineering Design*. PhD thesis, Michigan State University, 1994.

[74] CFI, "Inter-Tool Communication Message Dictionary," Tech. Rep. Document Number ITC-90-G-04, CAD Framework Initiative, Inc, 1991.

[75] CFI, "Tool Encapsulation Specification Standard," Tech. Rep. Document Number DMM-91-G-1, CAD Framework Initiative, Inc, 1991.

[76] K. W. Fiduk, S. Kleinfeldt, M. Kosarchyn, and E. B. Perez, "Design Methodology Management - A CAD Framework Initiative perspective," in *27th ACM/IEEE Design Automation Conference*, pp. 278–283, 1990.

[77] T. J. Scallan, "CAD Framework Initiative - a user perspective," in *29th ACM/IEEE Design Automation Conference*, pp. 672–675, 1992.

[78] W. Reisig, Ed., *A Primer in Petri Net Design*. Springer-Verlag, 1992.

[79] L. Ferrarini, "An Incremental Approach to Logic Controller Design with Petri Nets," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, pp. 461–473, May/June 1992.

[80] M. Zhou, F. DiCesare, and A. A. Desrochers, "A Hybrid Methodology for Synthesis of Petri Net Models for Manufacturing Systems," *IEEE Transactions on Robotics and Automation*, vol. 8, pp. 350–361, June 1992.

[81] J. Kljaich, Jr., B. T. Smith, and A. S. Wojcik, "Formal Verification of Fault Tolerance Using Theorem-Proving Techniques," *IEEE Transactions on Computers*, vol. 38, pp. 366–376, March 1989.