The Synthesis of Control-Dominated Application Specific Integrated Circuits Using Global Based Design Management

BY

Sally Hayati

CEng Technical Report 90-32

Electrical Engineering Systems

University of Southern California

Los Angeles, CA. 90089-0781

The Synthesis of Control-Dominated Application Specific Integrated Circuits Using Goal Based Design Management

by
Sally Hayati

A Dissertation Presented to the
Faculty of the Graduate School
University of Southern California
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

November 1990

ACKNOWLEDGEMENT

I thank my advisor Professor Alice Parker for her continued support through the years, and especially for understanding my need to work part time after my baby was born.

I thank the members of my committee, Professors M. Breuer and B. Abramson for their valuable criticisms and suggestions.

I thank my friends and former fellow graduate students Mitch Mlinar and Rajiv Jain for hours of discussions that helped me to maintain my sanity through it all. I also thank all the graduate students that I've shared an office with at USC, past and present, for their friendship and technical discussions.

I especially thank my husband Samad for his support and patience through the years of graduate school, and for sharing the responsibility of raising our family. I thank my little girls Megan and Claire for accepting so naturally that mommys go to school, too.

I gratefully acknowledge receipt of a three year Army Research Office Fellowship and financial support received from the Defense Advanced Research Projects Agency (contract number N00014-87-K-0861 monitored by the Office of Naval Research) . The views and conclusions contained in this thesis are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Contents

List Of Figures

Chapter 1

Introduction

1.1 Background

The automatic design of digital circuits is a goal pursued at many levels, from the physical level of device geometry and layout, up to the functional level of high level components. The highest level of automatic digital design is termed *synthesis*, defined as the creation of an implementation from a specification of the system behavior which indicates the "what" but not the "how" of system performance. Architectural synthesis systems create a register-transfer level (RTL) design from an algorithmic behavioral description. The elements of an RTL data path are operators that manipulate data, such as ALU's, adders, and multipliers, registers to store values, multiplexers and wiring to form the proper interconnections; the controller is also an RTL level component of the design.

The behavioral description is typically translated from a hardware descriptive language (HDL) specification to a data flow/control flow graph indicating system inputs, operations performed, control flow and data dependencies, and outputs. The user can place additional limitations on the implementation, such as an overall performance requirement, additional timing constraints, or area limitation. Steps of the synthesis task include:

1. Module selection: Abstract operations can often be implemented by several different operator or module types. Module selection determines which of these possible module types will be chosen for the current design problem. Commonly, a single module type is chosen for each operation type present in the specification.

2. Operation scheduling: Synchronous digital designs perform operations in distinct time steps called clock periods. During operation scheduling the design system chooses the total number of clock periods in which the specified behavior is executed and assigns operations to specific steps. A non-pipelined design with fewer time steps is faster but generally more expensive since more operations are scheduled in each clock period. Operations of the same type scheduled in the same clock period require different modules, whereas identical operations in different clock periods can be assigned to the same module.

3. Module allocation and binding: Once operations are scheduled, the exact number of each module type required is determined, and operations in the specification are bound to particular modules. Register and multiplexer binding must also be performed.

4. Control synthesis: Data path synthesis determines the values and timing of control signals required to regulate data path functioning. This information is required to synthesize the controller.

Many synthesis steps, such as operation scheduling, are known to have a complexity exponential to the size of the problem. An added complication is that the design steps and subtasks within each step are only partially independent.

This research is a part of the ADAM (Advanced Design AutoMation) System at the University of Southern California that includes the data path synthesizers MAHA, Sehwa, and MABAL, in addition to estimators, including PLEST and PASTA, which determine the characteristics of design implementations from high-level representations. More information on ADAM can be found in [JKMP89].

1.2 Problem Statement

1.2.1 Synthesis and ASICs

Application specific integrated circuits (ASICs) are designed and optimized for specific applications to achieve better performance and/or lower chip count, lower power consumption, and greater reliability than would be possible using

"off the shelf" LSI/MSI devices or general purpose microprocessors. The low cost and high density and speed of CMOS circuits, plus computer-aided engineering workstations have led to greater development of these devices and less reliance on standard parts. Because of the design complexity, however, ASICs are most commonly used to implement systems with high performance requirements, or for systems to be put into larger scale production. Systems with low manufacturing volume have a higher design cost per unit; the greater the savings in design time permitted by design automation, therefore, the greater the number of applications which can profit from ASIC technology.

Design techniques used for these circuits include *custom, standard cell, logic synthesis, silicon compilation* and *architectural synthesis.* Even custom or manual designs generally employ automatic design tools to some degree, especially placement and routing tools. Of all the design techniques, architectural synthesis has the greatest potential for reducing ASIC design cost because a greater range of tasks are automated, starting at an earlier stage of the design. This reduces the amount of manual effort involved and also permits systems designers, rather than chip experts, to create implementations.

One barrier to the increased use of architectural synthesis in ASIC design is the traditional synthesis focus on data path applications. These traditional applications are characterized by (1) simple control, (2) the dominance of arithmetic operations, and (3) limited interaction with the environment. In contrast, examples of common ASIC applications include interface functions such as protocol transducers, which allow independent systems to interact, I/O processors, and various types of glue logic. In addition, as greater chip density allows more functions to be placed on a single chip it becomes possible and desirable in some cases to combine interface functions with application specific processing on the same chip. Multiple examples of possible applications of this type exist within graphics terminals and digital television, in which large amounts of data are moved from input through processing stages and then to the screen.

A survey of many existing ASIC designs [Keu89] found that they are characterized by (1) control domination, (2) little arithmetic, (3) complex interfaces, and (4) modest speed/area requirements. The control function is a significant factor in the design for both interface circuits and processors bound by real-time

constraints. Such applications result in what are termed control-dominated ASICs. This thesis addresses the need to create new techniques for the automatic synthesis of control dominated ASICs.

1.2.2 Conventional Synthesis Techniques

To reiterate, more interfacing and real-time applications could be economically implemented as ASICs if design tools were available to decrease the development cost and time to market. The usefulness of high level design tools is nevertheless limited by a traditional focus on data path or arithmetic functions that operate in relative independence from the environment. When the design specification has complex control and interfacing functions and relatively less data manipulation, there are several problems with conventional synthesis methods. First and most basic, the function of the controller is too restricted. Behavior is implemented by assigning data manipulating operators to operations, and values to registers. The behavior of the control is limited to creating the proper sequencing of and conditions for the execution of data path operators.

For control-dominated applications this focus on implementing behavior using data path techniques is too narrow. The behavioral specification of a process constrained by real-time and interface requirements contains functions which are most naturally implemented using controller synthesis techniques, in addition to processing tasks which require data path implementation techniques. If interface functions are implemented using data path techniques, then interface input and output signals are latched into data path registers, comparison operators are assigned in the data path to determine the values of input signals, and constants involved in interface operations are stored in data path registers. This is particularly inefficient since most values, with the exception of addresses and interrupt vectors, are binary signals and constants.

As a consequence, interfacing behavior such as handshaking and interrupt processing is typically not implemented by high-level synthesis tools, as described in Related Research, Chapter 2. To quickly summarize here, Janus [BK87] synthesizes interface circuits but no data path functions can be included. An asynchronous circuit is produced using fine-tuned delay elements. The data path synthesizers

ISYN [NT86, Nes87] and ELF [Gir84, GK84] schedule operations based on detailed timing constraints imposed on data path values, but no control unit is synthesized. Interface behavior related to handshaking, bus protocols, and interrupt processing is not implemented, and issues related to synchronous interface behavior are not addressed.

A further difficulty with conventional synthesis techniques is that complex control functions cannot be handled. System correctness for interfacing and real-time processors depends not only on logical results but on their timing as well; time constraints may be multiple and both minimum and maximum.[1.1] Only a few design automation systems, including HERCULES [DMK88], ISYN [NT86], and ELF [Gir84] allow multiple constraints on the timing of input/output behavior. In addition, although synthesis systems usually create synchronous designs, most current systems do not deal with synchronous I/O signals, clock cycles and edges in the input specification, nor can they handle inputs with indeterminate (asynchronous) timing. Furthermore, the concept of parallel execution is related solely to data dependency considerations, and not to control flow.

Conventional synthesis approaches use inadequate design management strategies for the design of control-dominated ASICs. Data path synthesis systems typically attempt to balance a single goal against a desire for optimization, for example, to make an implementation as fast as possible within an area constraint, or as small as possible within a performance requirement. Decisions in scheduling and allocation all relate to these objectives. The existence of multiple minimum and maximum time constraints along with complex control including internal loops and parallel branches introduces new considerations and new types of interactions between design tasks.

To illustrate the problems of using conventional synthesis techniques to design control-dominated ASICs, consider a process which performs actions A or B, depending on two sequential binary values on input line X which is synchronized to a known clock, and places a response on line Z depending on the result. The first difficulty encountered is the synchronism of X; such behavior is difficult to convey using a data flow graph. In addition, data path synthesizers that do not address

[1.1] The term **interfacing processor** will be used to indicate an application with significant interfacing and processing tasks.

interface issues assume that the internal clock period can be set without regard for the arrival times of system inputs, which are expected to be in registers at the start of processing. A second difficultly lies in the treatment of input X and output Z. Input and output values are stored in registers with the appropriate number of bits, and constants may also be handled in this way. This treatment is inefficient for single-bit inputs and outputs like X and Z and the constants 0 and 1. Data path synthesis will assign a comparator to determine the value of X, with inputs from the registers containing X and the constants it is compared to. The output of the comparator is input to a multiplexer which controls the execution of A and B.

A control implementation of the same behavior can also be created. X is input directly to the controller; a conditional output from the controller determines whether A or B is executed based on the value of X; Z is produced directly by the control. Scheduling and module binding for data path operations contained within A and B are still performed using data path techniques, however. As can be seen from this small example, it is desirable for a synthesis system to have access to both control and data path synthesis techniques to implement the specified behavior.

1.2.3 Research Statement

The goal of this research is the development of high-level synthesis techniques for applications with complex control and significant interfacing functions in addition to data manipulation. Representation, design management, and synthesis issues relevant to the automatic design of control-dominated ASICs are addressed.

High level synthesis has traditionally focussed on more limited types of specifications. For this reason, the first goal of the representation work was to identify the range of tasks commonly implemented using ASIC technology so that the synthesis system could be designed accordingly. The representation languages available for high-level synthesis reflect the types of behavior handled by traditional systems. In addition, therefore, a second goal was to develop methods of representing the desired behavior using a Hardware Description Language (HDL) for user specification purposes. A third goal was to define a translation between the Hardware Descriptive Language specification and a graphical data structure for internal use

by the synthesis system. The translation is currently performed manually because of the large programming effort that would be necessary to automate it. In the representation effort a systematic procedure has been followed of identifying behaviors, defining the HDL description, and specifying a corresponding representation using the internal data structure. No attempt has been made, however, at formal proofs of correctness or completeness. An existing HDL (SLIDE) and data structure (DDS) were used and modified, as described in the problem approach section of this chapter.

Control synthesis techniques have been developed and implemented that work in conjunction with existing data path methods to automatically synthesize control-dominated ASICs. A design automation system called CONSPEC (CONtrol SPECification) has been created that performs control specification using the DDS Control and Timing Model graph as an input specification. CONSPEC uses a design management technique developed in response to the interactions and conflicts that arise in the presence of multiple time constraints and complex control. CONSPEC is the primary result of this research in addition to the representation work.

1.3 Thesis Overview

In the following sections we describe the design approach used in this thesis and introduce a flowchart showing the synthesis steps and systems that create the implementation. Then in Section 1.4.3 we introduce CONSPEC in more detail. In Chapter 2 we discuss the related work of several researchers from the fields of artificial intelligence and design automation. Representation issues related to control-dominated ASIC behavior, the system input specification, and the internal representation strategy are described in Chapter 3. The overall architecture and goal-based reasoning techniques of CONSPEC are described in Chapter 4. Chapter 5 explains the synthesis procedures used to perform control specification. Section 5.2 describes a procedure that derives a state table from the Control and Timing specification of the DDS. Section 5.3 describes procedures that can modify the state table to realize three explicit goals: parallel branch merging and minimum and maximum time constraint satisfaction. Chapter 6 presents conclusions and future research.

1.4 Problem Approach

1.4.1 Algorithmic State Machine Model

We base the design strategy for control-dominated ASICs on the Algorithmic State Machine (ASM) methodology [Com84, WP80]. Figure 1.1 shows a diagram of the model that consists of two interacting modules, the **control** and the **architecture**. The architecture is implemented by data path techniques; the controller is implemented using synchronous finite state machine (FSM) techniques. Inputs from the external environment can be directed to both modules, and outputs to the external environment may be produced by either module. The control also governs the behavior of the architecture, so values from the data path are passed to the control, which sends control signals to the architecture.

State machine techniques are useful in creating many different types of circuit, avoid problems such as output glitches and race conditions, and can be used to create optimal sequential circuits through minimization of the number of flip-flops and other circuitry. Not all optimizations need to be applied, depending on implementation requirements.

Control styles for finite state machines range from random logic to PLA matrices to microcode; different implementation styles will lead to controllers with different performance/cost characteristics. The reliability, testability, and efficiency of circuits produced using synchronous FSM methods makes them more preferred in industry over asynchronous control circuits, except for applications whose performance requirements are too great for synchronous designs, or whose size complicates clock distribution.

Two types of state machine can be defined: the **Moore** machine whose outputs depend only on state and the **Mealy** machine whose outputs depend not only on state, but possibly on inputs as well. The approach used here is that of the Mealy machine. **Conditional outputs** are asserted during a state only under the right controller input conditions, whereas **unconditional outputs** are produced during particular states regardless of the value of any controller inputs.

State transitions can depend on input conditions as well as the current state, and therefore can also be one of two kinds. A conditional state transition exists when the transition from one state to another occurs only under certain input

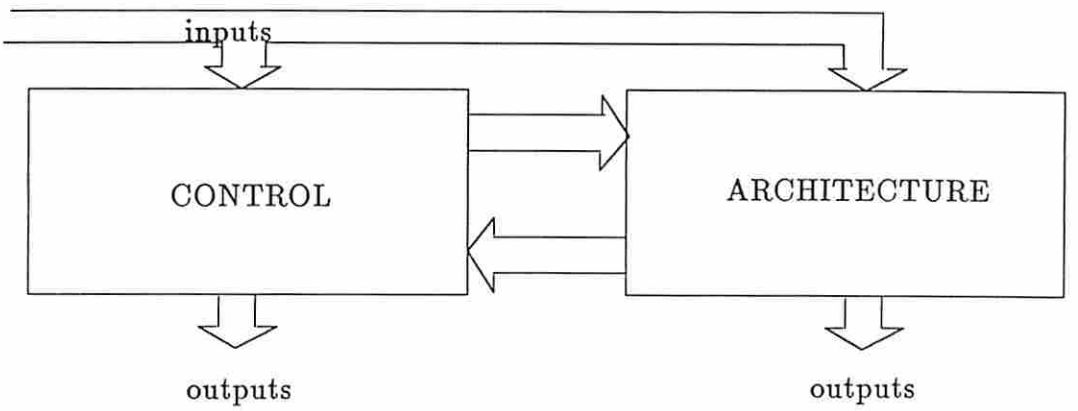


Figure 1.1: Algorithmic State Machine Model

conditions. In that case there will be more than one possible next state transition. A state with an unconditional transition is always followed by the same next state, regardless of controller input values.

Information on the timing and values of data path control signals must be incorporated into the behavior of the controller. For this reason, data path implementation precedes control design.

1.4.2 Control-Dominated ASIC Synthesis

Figure 1.2 shows a diagram of the architecture for a proposed ADAM subsystem for the synthesis of application specific integrated circuits to implement the ASM methodology described in the previous section.

1.4.2.1 Representation Issues

1.4.2.1.1 Behavior of Control-Dominated ASICs The digital design literature was analyzed to determine the characteristics of control-dominated ASICs, thereby determining the types of behavior that must be represented and synthesized by the design system.[1.2] Based on this research, a behavioral categorization was established for interfacing processors. The behavioral categorization is continued within the context of the specification language as well as the internal representation used by the synthesis system. This systematic approach to the

[1.2]Sources such as the IEEE *Transactions on Consumer Electronics* and *Computer Design Magazine* were consulted.

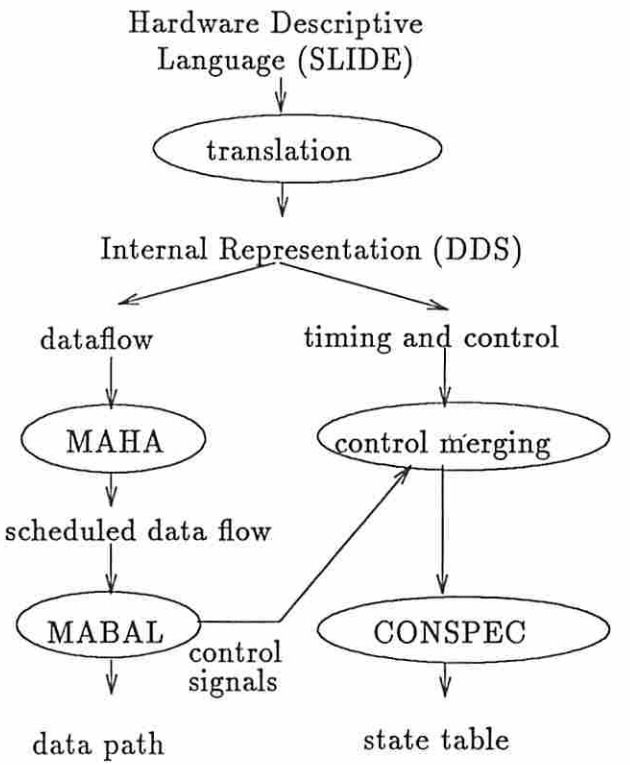


Figure 1.2: The ADAM Subsystem for the Synthesis of Application Specific IC's

representation of control-dominated applications increases the generality and usefulness of the synthesis system that works from the representation. This behavioral categorization is described in Chapter 3.

The categorization effort is similar to that found in [Par75]. The goal of that work was development of a general purpose programmable interface, whereas this work focuses on the synthesis of circuits for special purpose applications.

1.4.2.1.2 Specification Language The specification is input to the system using a hardware descriptive language that allows behavioral specification without forcing the user to commit to implementation details. This representation should be capable of describing data manipulation, interface behavior, complex control flow, and detailed timing information. It must be capable of defining the lines, registers, or clocks that are used at the design interface.

There are many hardware descriptive languages available. The SLIDE [PW81] hardware descriptive language has all the capabilities of ISPS and the syntax is

similar.[1.3] In addition, SLIDE was designed specifically for the description of I/O protocols, interfaces, and interconnected systems. The SLIDE simulator allows I/O simulation and verification of synchronization mechanisms. Synchronous as well as asynchronous events can be represented, as can some useful control constructs such as delays and time-outs. This capability is in addition to the standard arithmetic and Boolean operations and control constructs such as IF, WHILE, and DO statements. SLIDE was therefore chosen because of its expressive power and the convenience of its familiar syntax.

Extensions and modifications have been made to SLIDE in those cases where its representation strategies are found to be incomplete or ambiguous in relation to the behavioral categories.

1.4.2.1.3 Internal Representation The first step in ASM design is to partition the high-level system behavior into architecture and control. The choice of the best behavioral partition depends on specification characteristics and on performance/cost constraints imposed on the implementation. Neither control not architecture is necessarily dominant; a design can consist of a large controller and small architecture, or vice versa. By contrast, in the conventional data path synthesis model the control is a secondary element of the implementation strategy compared to the architecture.

In Figure 1.2 the action marked *translation* corresponds to this step, in which the HDL specification is translated to a graphical internal representation. The Design Data Structure (DDS) developed at USC by J. Granacki, D. Knapp, and A. Parker [Gra86, KP83] can support the ASM view of design because high level behavior is represented using two models, the Data Flow and the Control and Timing. The Data Flow Model represents data manipulating operations and defines the data dependencies between them, showing maximum parallelism. The Control and Timing Model includes conditional and parallel branches, time constraints, loops, sequencing requirements, and interface behavior such as protocol interpretation. Additional design information is contained in the Structural and Physical Models. In the DDS, all relations between the Models are explicitly specified by

[1.3] At the time this work was begun VHDL was not as widely accepted as it is now; ISPS was far more commonly used in the design automation community.

bindings. Our paper reprinted in Appendix A [HPG88] gives an overview that will allow the reader unfamiliar with the DDS to understand the examples in this thesis.

Until now, the Control and Timing Model had not been used for synthesis purposes; the data path synthesis systems of the ADAM system [PPM86, PP86] use the Data Flow Model. Granacki defines the basic structure and use of the Control and Timing Model [Gra86], but does not give details on representations for many behaviors found in control-dominated ASICs, since his work focused on system-level considerations. For that reason some new conventions and minor modifications to the Control and Timing Model are established for the representation of behaviors that include process initiation and process priorities, signal events, maximum and minimum time constraints, delays and timeouts, and both synchronous and asynchronous events.

The translation from SLIDE to DDS is currently performed manually, however the behavioral categorization provides a foundation for the translation.[1.4] During this translation a unique partition is created between the data flow behavior, corresponding to the architecture, and the control and timing behavior, corresponding to the control. This partition is not present in the behavioral SLIDE description. For a single SLIDE specification multiple partitions could be created; we have therefore defined a canonical translation method that will result in a satisfactory implementation for most specifications. In Chapter 6 on future research we describe methods that could be developed to effect different partitions through transformations to the DDS representation.

The formal proof of the equivalence of different specifications is a topic of current research and there is no accepted or clearly superior method available. The focus of this research is on synthesis methods; for this reason the behavioral equivalence of SLIDE and DDS representations is based on a careful but informal semantic analysis of SLIDE and DDS. Also because of the difficult research issues involved, the equivalence of the DDS specification and final implementation is not proven, but is based on conventions established in Chapters 3 and 5 and verified by manual design.

[1.4]The automation of this task would be a large but conceptually straightforward programming task, since a translator now exists from a VHDL subset to the DDS.

Behavior represented in the Data Flow Model is realized using data path synthesis techniques, and the behavior in the Control and Timing Model by state machine techniques.

1.4.2.2 Data Path Synthesis

Data path synthesis is performed before control synthesis. Existing systems such as MAHA are used to schedule the data flow operations.[1.5] Module allocation and binding are performed by an existing system called MABAL [KP], thus completing data path design. From the output of MABAL the values and timing of data path control signals are manually determined and incorporated into the graph of the Control and Timing Model. This step, marked *transformations* in Figure 1.2, is not automated. A description of the method used is given at the beginning of Section 3.2.4.3.

1.4.2.3 Control Synthesis

Following data path synthesis, control synthesis is performed from the Control and Timing Model. This step is done by the design automation system called CONSPEC (CONtrol SPECification). A major goal in the development of CONSPEC was flexibility in controller implementation style, rather than commitment to a single style such as microprogramming or PLA. An associated goal was the use of existing control and logic optimization techniques. To achieve these goals, a Mealy model FSM specification is produced in the form of a state table listing the set of states, outputs, and state transitions under all input conditions.

A synchronous finite state machine can be synthesized from the state table. This involves state reduction and assignment, creation of a circuit to cycle through the correct sequence of states, and the design of input and output forming logic, and can be performed by a number of existing systems for state machine and logic synthesis and optimization [Peg88, MSVV83, GR87, LL84]. The interface between CONSPEC and these systems has not been automated.

[1.5] Some modifications to SEHWA would be required to account for complexities such as multiple time constraints and internal loops.

1.4.3 CONSPEC

All synthesis techniques described in this thesis have been implemented in a computer program called CONSPEC that is written in Prolog. CONSPEC accepts a high-level behavioral description in the form of a DDS Control and Timing graph that includes signal values that control the data path and timing as well as interface behavior. From this graph CONSPEC derives a specification for a Mealy model finite state machine in the form of a state table.

The control specification involves three general tasks. First, CONSPEC determines which behaviors described in the graph should be included in a single state and the number of states required to implement the behavior. Second, it decides which outputs should be produced during particular states. Third, it determines which, if any, input values affect the conditions under which outputs are produced and under which next state transitions occur for each state. The three tasks are performed one state at a time. After derivation of the state table, time constraint satisfaction is performed, with possible modifications to the table. Optimizations to the state table can also be made at this point.

Because control-dominated ASICs are characterized by multiple minimum and maximum time constraints it is possible for conflicts and interactions to occur between individual decisions. Unless the design system is aware of such interactions, the design may be adversely affected. As an example, a local minimum time constraint may be nested within a global maximum time constraint. If the minimum constraint is satisfied by adding delays the effects on maximum constraint satisfaction should be considered and possible adjustments made, otherwise the maximum constraint may be inadvertently and unknowingly violated.

A knowledge of the effects of interactions is useful not only in avoiding problems, but in detecting problems that cannot be avoided and reporting them to the user. In other cases, positive interactions can be taken advantage of to improve the design; an example of this is nested constraints that are either both minimum or maximum.

A goal-based design management approach was developed for CONSPEC to take into account these and other types of interactions between design tasks. Robert Wilensky's planning strategies developed for natural language understanding [Wil83] were particularly useful in this development, as described in Chapter 4.

Chapter 2

Related Research

In this chapter we first describe four systems that have extended synthesis to specifications with multiple timing constraints and interface primitives. Three of the systems approach the problem from a data path perspective, and one from an asynchronous control perspective.

Next, a number of planners are described. There are many similarities in the functions of design and planning. Planning is a problem solving technique that maps out a sequence of steps to achieve the system goal or goals without actually executing any of the steps. In this way difficulties can be anticipated and wasteful actions avoided. Failure to plan ahead can even preclude the achievement of certain goals in the case of interactions, resource limitations, or a hostile environment. Correspondingly, for design the goal is the creation of a plan for the construction of some artifact. The existence of a design plan saves expensive trial-and-error using actual physical components. This similarity in function between design and planning suggests the possibility of using common methods for both.

There are differences as well, however, between planning and design. A planner produces an ordered or partially ordered sequence of the operators available to it; each operator generally has some real world procedural equivalent, e.g., go to point x, which may have been simulated, but not executed. Alternately, the output of a design system is not a sequence of the system operators, but the result of executing some sequence of system operators. The resulting configuration must be analyzed or simulated to be certain that it meets specifications. The final result of the design process is a plan to be carried out in the real world, although that plan is typically not a representation of actions to be performed, but a representation of

a physical object to be constructed. For these reasons there is a more complex relationship between input specification, operators, and system goals for design than for general planning tasks.

2.1 Elf

Emil Girczyc created the first synthesis system geared to handling designs that have significant interaction with their environment [Gir84]. His system, Elf, accepts multiple timing constraints rather than the single, overall performance constraint more commonly handled by data path synthesizers. System specifications in the ADA programming language are translated to a control/data-flow graph (CDFG) for internal use. Port operations are represented in this graph by the operations GET and PUT. Timing constraints are initially specified by a list of constraints between labelled points in the CDFG. Absolute timing constraints unrelated to data dependencies are represented by a different type of arc in the CDFG. Girczyc employs a user supplied table to estimate the timing of control operations, and incorporates a measure of control complexity in the cost estimates by noting the number of module control signals. The control cost of loop nodes is determined by fixing the number of iterations: indeterminate loops cannot be handled as such.

Satisfaction of timing constraints is achieved through use of a modified urgency scheduling technique[The76]. The system specification given by the user establishes a clock period, and synthesis proceeds one *time slot* at a time. Each time slot will have one or more operations from the data flow graph assigned to be performed during that interval. Only *feasible* nodes may be considered for each slot, meaning that all data dependencies have been satisfied. Data flow nodes are assigned *urgency weights* based on data dependency and timing constraints. Nodes may have a range of time slots in which they are feasible, within which neither data dependencies nor timing constraints are violated. To schedule these nodes, Elf considers the node's weight in conjunction with a measure of the cost of realizing the node in hardware. Nodes that can share already assigned hardware will have a cost associated only with interconnect. The more urgent the node, the higher the cost of allocation that will be allowed; nodes with low allocation costs are likely to be assigned regardless of urgency. Elf thus performs module assignment

in conjunction with scheduling, both of which are influenced by the desirability of resource sharing.

One interesting feature of Elf is the use of graph transformations to modify the form of the specification and thus create a different implementation. Using the *transformational approach*, a translator identifies the behavior in the HDL specification and translates it to a corresponding internal graph form. The resulting configuration can be considered a preliminary configuration that may be manipulated by the system to better meet system requirements. The system manipulates the internal representation using a set of predefined transformations. Graph matching must be performed to determine the applicability of a transformation, and graph manipulation must be performed to apply the transformation.

An approach like Girczyc's could potentially be used for data path synthesis in conjunction with CONSPEC for control synthesis, particularly for specifications that include a clock period.

2.2 ISYN

A later system with capabilities similar to Elf is ISYN (Interface SYNthesis) [NT86] [Nes87], developed by John Nestor at CMU. This system designs data paths with multiple maximum and minimum time constraints and I/O events. The internal data structure is a modified version of the value trace (VT) [McF78], which is a data flow graph that incorporates control information. Nestor has added I/O operations plus timing constraints to the VT. Structural information about interface ports, including bit width and electrical characteristics, is contained in a separate file.

Two port operations and one control operation are represented as operations in the data flow graph. The port operations are INPUTP(port), which returns the value read from "port", and OUTPUTP(port,value); these correspond to the GET and PUT operations used in Elf. OUTPUTP has special values *TS DISABLE* and *ENABLE* to describe tristate outputs. Timing of the port operations is fixed in relation to the clock cycle. The input operation executes at the beginning of a control step; the output operation completes execution at the end of a control step.

One control operation, termed TNEXT, was introduced to specify absolute sequencing information. This was necessary since data dependency alone could determine the sequencing of operations in the original VT, however, interface events often have sequencing requirements that are free of data dependencies. This operation performs the same function as the absolute timing arcs in Elf's CDFG. Representation of timing constraints is achieved by affixing a label to each data flow operation involved in a constraint, and listing all maximum or minimum timing constraints between operations by referring to their labels. The system specification thus includes a file listing all the timing constraints associated with the VT, in addition to the VT itself and a structural specification detailing port characteristics.

In Nestor's approach a clock period is predetermined and operations are scheduled for execution one time slot at a time until all are scheduled. The definition of scheduling feasibility, however, is extended to include minimum timing constraints in addition to data readiness. Although Girczyc mentions minimum timing constraints, his system does not seem to consider them when weighing nodes. The range of time slots in which an operation is considered feasible by ISYN is thus limited on one end by data readiness and minimum timing constraints and on the other by maximum timing constraints. ISYN schedules all nodes for execution as soon as they are feasible. This means that scheduling is performed with the single goal of achieving timing constraints, and the possibility of sharing operators and reducing cost is overlooked. The system cannot handle loops, with or without limits on the number of iterations. Chained operations are supported (a sequence of operations within a single time step), as in Elf, and in addition operations may be scheduled over multiple time steps if their execution time exceeds the length of the clock period.

The output of Nestor's system is a list of the scheduled operations. A modified version of the EMUCS data path synthesizer [McF81] then allocates hardware for any data flow operations present. The port operations are prebound to some hardware templates that can be connected to the data path and controller, once it is synthesized. Variables in the specification that are initialized to a constant and are inputs to only increment, decrement, or shift operations may be assigned to counters or shift registers by a separate hardware allocator.

2.3 Janus

G. Borriello at UCB also produced an interface synthesis system. *Janus* accepts two interface specifications from which it produces the interconnecting transducer circuitry. The internal data structure of this system is termed an *event graph* [BK87], which is a timing graph derived from a conventional timing diagram. This is a different approach from the control/data flow graph methods described above. Event graph nodes correspond to transitions in signal logic levels, with two node types indicating input and output. Simultaneous events are combined in a single node. There are two arc types: the first indicates sequencing constraints, the second signifies timing constraints between two events. Conditionals are represented implicitly by the connections between events. Signals may be described by Boolean expressions if that is more natural than the timing diagram approach. Synchrony is specified by identifying each node as either asynchronous or synchronous, with a clock specification in the latter case.

Each node, or signal event, is implemented using an S-R latch. These are interconnected with circuitry chosen by matching hardware templates to arcs of the event graph. There are three basic templates that correspond to various combinations of the two types of events, synchronous or asynchronous. Each template has an and-gate with one input to accept signals that condition the event. After all templates are chosen, delay elements or clocked flip flops are added to meet minimum timing constraints. Loops require some additional circuitry, primarily a counter. The final design is asynchronous: signals ripple through these circuits from inputs to outputs. There is no data path design.

2.4 CAMAD

CAMAD (Computer Aided Modeling, Analysis, and Design of VLSI systems) is a synthesis program created at Linkoping University, Sweden, by Zebo Peng. The internal design representation consists of a data path graph and an extended Petri net graph. The data path graph is a structural diagram whose nodes represent data path elements such as registers, busses, and operators, and whose arcs model the connections between these elements. The sharing of operators is represented by

merging multiple nodes into a single node and redirecting the register connections. The synthesis process therefore proceeds from an initial structure that reflects the maximum parallelism and a Petri net whose transitions represent control signals that enable register transfers in the structure. During synthesis additional registers may be added for intermediate values of the specification, and the number of operators may be reduced by sharing. External timing constraints are recognized by maintaining a partial ordering relation between input and output events provided by the specification, in a fashion reminiscent of McFarland's Behavior Expressions [MP83]. Absolute timing constraints are not considered. The control is implemented by generating a microprogram from the control Petri net.

2.5 STRIPS

STRIPS introduced a solution to the representational problems of planners that has been used since by virtually every domain independent planner. The state of the planning world, or the *world model*, is represented as a list of assertions. Operators delineating the actions available to the planner are each represented by three lists of assertions. An operator is considered to be applicable if the assertions contained in its *preconditions* are all present in the world model. The effects on the world model of applying each operator are represented by an *add list*, whose assertions are added to the world model if not present already, and a *delete list*, whose assertions are removed from the world model if present.

A supposition used in planning that has come to be called the *STRIPS assumption* is that all effects of operators that are relevant to the planning task may be so represented. Any effects not explicitly represented, called side effects, should either be logically derivable from axioms of the system (e.g., if you move *object a* upon which sits *object b* from x to y, *object b* is then at y, not x) or irrelevant to the planner's task. If this assumption is violated, the planner cannot be guaranteed to create a correct plan, let alone an efficient one.

2.6 Hacker

Hacker was designed to perform conjunctive goal satisfaction while considering various types of interactions that might occur between the goals that would invalidate the plan. Hacker is a *linear planner*, meaning that the system imposes a strict time ordering on the achievement of subgoals. The subgoals are initially assumed to be independent. To deal with situations in which the assumption proves to be false, Hacker has an ability to debug plans by detecting certain kinds of interactions.

Hacker's operator choice and representation strategies are similar to STRIPS. It is primarily concerned with detecting conflicts due to the linear ordering chosen for goals. It cannot, for example, reconsider operator choice for some combination of subgoals. The types of fixes it is therefore capable of performing consist of the insertion of new operators to achieve missing prerequisites and time reordering of operators. It has a capacity to represent primitive resource consumption, such as the amount of room left on top of a block, because the strict ordering of operations leave no doubt as to the state of the world at each point.

A contribution that Hacker made is the concept of debugging a plan using knowledge of potential interactions that is kept separate from representations of the operators themselves. Maintaining knowledge of interactions separate from operator representation keeps the operators simple, and bug detection and fixes can reduce computation time compared to simpler combinatorial problems. However, the Hacker approach assumes that interactions are due to improper goal orderings. In digital design, the existence of interaction may be due not only to improper goal ordering, but to improper operator choice or application.

2.7 NOAH

NOAH incorporates many of the methods and representations used by STRIPS and a feature similar to the bug detection of Hacker into a more advanced planning strategy. It introduces the concept of hierarchy into planning and incorporates procedural as well as declarative knowledge. These developments are obviously relevant to the needs of design, which is inherently hierarchical and procedural. The plan representation of NOAH is termed the *procedural net.* This structure

represents goals *nonlinearly*, meaning that the operators are arranged without a commitment to time ordering. A partial ordering is imposed only if stipulated by a plan critic, a procedure designed to detect and avoid interactions. Critics are different from bug detectors in that they do not debug incorrect plans, but constrain the ordering of subgoals so the resulting plan will never contain interactions. Of course, this reflection on the nature of critics assumes that the only interactions that could result in incorrect plans are those resulting from improper ordering.

At the starting point of NOAH's planning process the procedural net consists of a single goal; this goal is then decomposed into subgoals by a domain specific procedure. These subgoals, unordered, take the parent goal's place in a new net. Critics then examine the net for potential interactions, taking corrective action if necessary. This decomposition into subgoals continues downwards until a level of goal is reached that has a primitive solution. This process is similar to that of hierarchical problem decomposition used in design.

One problem with NOAH's representation is that each high level goal is expected to specify a single decomposing operator for each of its subgoals. If instead, several possible decomposing operators are recommended, a parallel plan is created for each alternative and expanded until one is deemed to be "simpler" than the others. Variable instantiation is delayed until there is but one viable candidate. This approach to planning would perpetuate the combinatorial problems of design.

NOAH's method of operator representation is similar to that of the previous two systems, as are the type of interactions detected. One type of positive goal interaction detection has been added: if multiple, identical goals are present in the plan then all but one are deleted, on the theory that if a thing is done once, it is done. Although this may be true for certain domains, deciding to share plans for multiple design goals requires a more sophisticated strategy.

2.8 MOLGEN

MOLGEN [Ste81] represents an important step in expanding the type of interactions considered during the planning process. NOAH constructs plans by initially underconstraining the operators in terms of ordering, but does commit to using specific operators, and therefore must carry along parallel plans for every possible

operator choice. MOLGEN plans in terms of *abstract operators* that may have several instantiations possible. Its initial planning steps underconstrain both operator and variable choice; this is called the *least-commitment* strategy. The use of this technique means that the planning agents looking for interactions can consider a variety of types and that "fixes" can go beyond the ordering of goals.

There are actually two major concepts involved in carrying out MOLGEN's planning strategy. The first is the use of hierarchy, not only in the domain representation, but in the planning process itself. The second is called *constraint propagation.* The theory is that interactions that occur at the more specific, lower levels of a plan will not occur if planning is conducted using sufficiently abstract steps. As planning progresses the need for certain operators or variables to conform to specific requirements, called constraints, becomes apparent. These constraints narrow the range of possible choices, as those options that cannot meet requirements are weeded out. Interactions between subproblems can be modeled because constraints on a specific operator or object discovered in the solution of one subproblem can be propagated to other subproblems that are also concerned with that entity. Commitment to instantiating an abstract entity with a specific operation or constant is delayed until no more constraints on the value can be determined. If there are no choices left that meet the various constraints, failure results. If there is only one option remaining the solution is found. If multiple choices remain a guess is made that may later need to be revoked.

Constraint propagation is a very general technique that can take several forms. The particular way it is applied in MOLGEN is not directly applicable to digital design because of the form of the constraints involved. Nevertheless, the general MOLGEN techniques for abstraction and constraint inference are similar to the common synthesis approach of initially designing with generic "register-transfer level" elements such as adders before progressing to module binding with specific physical modules. We will not consider Stefik's approach to meta-planning and the nature of the hierarchy used in detail, but now look more closely at constraint propagation.

Constraint propagation makes use of a structure termed a *constraint network.* Each node of the network represents an entity with some value, such as an antibiotic or bacteria in MOLGEN, and may be connected by constraints to other nodes,

thereby restricting the entity's value by creating relations such as "bacteria x should be resistance to antibiotic y." This type of constraint propagation is called *label inference*, in which nodes are labelled with the set of all possible values. Locally specified constraints are propagated throughout the network by eliminating all node set values that do not meet the constraints.

To apply constraint propagation to digital design, nodes could represent design elements and node labels could represent the characteristics of different implementations of that element. For example, nodes could be data path operations, such as addition, and each node label could describe an operator to implement the operation, such as a carry-look-ahead or ripple-carry adder. Node labelling is complicated by the need to designate the values of more than one parameter for each candidate operator, such as area, delay, and power. An additional complication in digital design is the possibility of sharing a single operator to implement different operations.

As a result of propagating all constraints, nodes labels will contain a restricted set of values. In general, it is not true that choosing any single value from each node set will give us a combination of choices that meet the constraints. The only thing that constraint propagation can guarantee is that any single value remaining in a node set can participate in some combination of choices that will meet the constraints. The resulting node labels do not therefore give us the answers to our problem, but provide a pruned search space within which we must use some additional problem solving strategy. MOLGEN's approach was to guess a value if no more constraints could be propagated and multiple values remained.

Unless certain simplifying factors exist, such as locally limited constraint propagation, a sufficient set of constraints to adequately limit node values, and simple forms of labels and constraints, constraint propagation can be very complex computationally and inadequate for query answering. In fact, in [Dav87] Davis states that, with the exception of problems of order relationships (such as NOAH deals with) and bounded differences, "We have not found any arguments that the partial results computed by label inference should be adequate for the purposes of AI." This seems to be the case for design with resource constraints, in which the simplifying factors do not hold.

2.9 TWEAK

TWEAK [Cha87] is an interesting system in that it claims to be the final word on domain-independent, non-linear, constraint-posting, conjunctive planning that can guarantee correctness of the plans produced. TWEAK, like all such planners, uses the STRIPS action representation, that Chapman proves to be both necessary to guarantee correctness, and critically limiting to the usefulness of domain-independent planners since it does not allow consideration of uncertainty or indirect or implied effects. He states, "...in retrospect all domain-independent conjunctive planners work the same way; that the action representation that they depend on is inadequate for the real-world planning; and that desirable extensions to the action representation make planning exponentially harder."

Chapman presents proofs of the correctness and completeness of TWEAK's algorithm, which he compares to a Turing machine, and attempts to prove that the planning task is undecidable if there are infinite initial situations or if actions with conditional or context-sensitive effects can be represented. Plannners such as SIPE [Wil84] that introduce derived or conditional effects into a nonlinear planner he shows to be incomplete and not generally correct (although not useless for that reason), and he also shows that linear planning, though more powerful in that respect, is exponentially less efficient than nonlinear. As his final conclusion, Chapman suggests that the focus on general techniques for domain-independent planners may be misdirected, and that the only solutions to the frame and representation problems powerful enough for real-world problems will be heuristic ones.

2.10 DONTE

DONTE [Ton88] is a knowledge-based system that performs circuit design using a goal-directed approach. DONTE specifications include a description of the physical structure and the functions it is to possess; in particular, it has been tailored for the design of stacks. The physical structure is described at a higher level, namely word and bit length for the stack. DONTE knows how to implement this at the next lower levels as the correct number of registers of the proper bit length,

and at the next level down as flip-flops, etc. Functional implementation is also determined by looking up the function, such as pop, and finding possible plans for its realization. The choice of a plan depends on other specification details, such as TTL versus other type of technology. Some level of interaction of goals is recognized by propagating functional constraints, for example, that the push operation of a stack should have the same plan type as that chosen for the pop operation. This is possible because the specification as a whole is considered a member of a predefined class, thus linking the different functions specified.

In a fashion similar to MOLGEN's approach, DONTE organizes circuit objects and functions into class hierarchies. For example, the stack operations *push* and *pop* are members of the *list processing* class of functions. The design agents of DONTE are an Advisor, a Planner, and a Designer: the structure is similar to that of NOAH described previously. The Advisor maintains a goal network. The initial goal is given in the specification; this is broken down into subgoals using a predefined "refinement operator" associated with the class the goal belongs to. There are three categories of goals: completeness, correctness, and resource limitation. *Bottleneck* goals are identified using a fixed goal priority scheme, in which resource limitation goals are highest priority, followed by completeness goals. Correctness goals are implicitly realized.

Only one resource goal can be specified, in the form of a number limit on particular design components, such as control lines. If this number is exceeded during design, a "patching" operation that knows how to handle such violations is sought. If not found, backtracking occurs, and the system attempts to find a different sequence of plan expanding operations. The Planner maintains the design plan, which defines a partial ordering on the goals. The Designer creates the implementation by executing or simulating the plans associated with the goal currently identified as the bottleneck.

2.11 The VLSI Design Automation Assistant

The VLSI Design Automation Assistant (DAA) is a knowledge-based expert system for high-level design synthesis created by Kowalski at Carnegie-Mellon University

[KT83, Kow84].[2.1] Expert design advice on MOS microcomputer implementation was incorporated in the system in the form of approximately 300 production rules. DAA completes the entire register transfer synthesis process in a single program. It deals with the large search space by applying a partitioning algorithm prior to design [McF83], thereby influencing the nature of the resource sharing carried out. Shared ALU's and buses are used whenever performance will not be affected.

The designs created by DAA were critiqued by expert designers, whose advice was used to add to the knowledge base and produce better implementations for designs that included the IBM-370. DAA was the first system to demonstrate the advantages of a pure knowlege-based approach to synthesis, particularly extensibility, flexibility, and understandability, as well as certain disadvantages, including lack of generality of the rule collection and speed limitations.

2.12 ULYSSES

Carnegie-Mellon University's ULYSSES system (Unified LaYout Specification and Simulation Environment for Silicon) [BD86] is a CAD tool integration environment. ULYSSES provides for the automatic invocation of translators (provided by the user) from other representations to the internal representation so that programs with arbitrary input and output formats may communicate with each other automatically. Each CAD tool known to ULYSSES is treated as a knowledge source that can be scheduled interactively with the user, but more typically is scheduled according to user-specified scripts. Scripts define control sequences of design tools referred to as design methodologies, that can be particularly useful for less experienced designers. New tools may be introduced by defining new translators and introducing new or modifying old scripts.

2.13 DPE

The ADAM Design Planning Engine (DPE) [GKP85] is an expert system that determines its own design control strategy using a planning strategy. The DPE

[2.1] Much of the material presented in this and the remaining sections of this chapter can be found in our paper [PH87].

builds a design plan by choosing from a set of possible analysis and synthesis tasks and tools, including clocking scheme synthesis, component selection, critical path location, area estimation, and hardware allocation. DPE represents plans as sets of partially ordered operators like those described by Sacerdoti [Sac77]. Design tasks are first selected based on characteristics of the problem specification, then added to the plan on the basis of their pre- and post-conditions and an estimate of the advisability of choosing that particular task. Once the plan is complete, the tasks are executed in the prescribed order.

Planning knowledge is contained in a collection of rule sets; digital design knowledge is found in data structures called *frames*. Frames consist of named slots that provide a context and organization for information about the object and may establish object hierarchies (such as is-a or part-of) and allow procedural attachment. Valid inputs and outputs for each tool are specied in the frames, and preconditions that must be valid prior to tool usage are maintained in the frames and checked by the planner. New tool frames can be added without any modification of the planning rules.

Chapter 3

Problem Representation

In Section 3.1 we categorize and enumerate the types of data flow, I/O, and control behavior that are characteristic of control dominated ASICs and that generally are not handled by conventional data path synthesis systems. This analysis is based on a study of the design literature[3.1] and descriptions of existing systems in manufacturers' publications. We then describe in Section 3.2 how this behavior can be specified for synthesis purposes using the Hardware Descriptive Language SLIDE [PW81]. The research contribution of this section involves modifications to the language as well as the proposal of conventions related to timing characteristics and other issues, as indicated in the text. For descriptive completeness, the section includes standard SLIDE constructs as well.

The last section sets forth a method to represent the behavior of Section 3.1 using the Design Data Structure (DDS). The research contribution of this section involves some modifications and the introduction of new conventions. Because the Control and Timing Model has been little used by synthesis systems, it was necessary to determine a standard method of representing complex behaviors such as process priority and searches for input bit patterns using sliding windows. One complication is that more than one way usually exists to represent a single high level behavior in the DDS, particularly when using both the data flow and timing graphs. Section 3.3 describes a canonical method to represent HDL constructs in the DDS. Some simplifications, as described in the section, are introduced to the DDS Timing representation for synthesis purposes.

[3.1] Magazines such as *Computer Design* and the IEEE *Transactions on Consumer Electronics* were particularly helpful.

3.1 Behavior of Interfacing Processors

This section gives a description of four categories of function, pointing out behaviors of particular significance to interfacing processors. The first category consists of behavior that requires *data manipulation.* The second covers *logic-level communication*: this encompasses I/O tasks at the digital logic level. The third category describes issues in *control flow* and the final category is *timing behavior.*

A fifth functional category, *signal-level communication*, will not be covered in this thesis. Signal-level communication covers technical and analog functions that deal with voltage levels and signal generation. Example functions in this category include data separation and signal filtering and enhancing.

The classification strategy defined here is comparable to that employed in [Par75]. Parker defines four categories of interfacing primitives: control, data input/output, data storage, and data manipulation. The data storage function is not considered separately in this work, but is contained within the logic-level communication and data manipulation categories instead. Similarly, Parker does not define a separate category for timing behavior but includes it with control.

3.1.1 Data Manipulation

Data manipulation is needed for the varied application tasks ASICs are called to perform. Even application tasks associated primarily with interfaces between devices involve data manipulation for non-trivial tasks. Error detection and correction circuitry performs operations such as cyclical redundancy and parity checks. Collision detection and address decoding require data operators. The more complex interface circuits acquire real-time data, using some communication scheme, and perform significant processing. For example, graphics applications include the translation of geometric objects in virtual coordinates to points in local coordinates to pixels on the display screen. Additionally, many ASICs integrate application circuitry along with circuitry dedicated to I/O tasks.

The types of operations found in interfacing processors therefore include all the standard operations used by traditional data path synthesizers such as addition,

multiplication, division, subtraction, and logic functions. There are some operations peculiar to interfacing processors, however, that are implemented by less conventional operators, including:

1. **Priority encoders and decoders** are used to implement priority arbitration schemes for tasks such as interrupt handling and process initiation.

2. **Bit manipulation** is frequently performed because of data formatting and decoding.

3. **Counters** are used for a variety of tasks ranging from event counting to random number generation.

4. **Shift registers** are especially important for synchronous applications in which sequential bits are shifted into a register to examine their value.

5. **Comparators** are used to compare device addresses with addresses on the bus, as well as device priorities with the current priority level, among other applications.

6. **Look-up tables** are commonly found in ASICs because of their special purpose nature. An example application is polynomial transformation.

Many of these structures are sequential in nature, in which output depends not only on present but on past inputs as well. Much of the data manipulating behavior common in interfacing processors is implemented using such operators whose behavior relies on state as well as inputs. A related aspect of data manipulation by interfacing processors is the comparison, averaging, integration, or other manipulation of a *sequence* of data values [FL86]. In addition, module operation often relies on clock edges, as in the case of shift registers and counters.

3.1.2 Logic-Level Communication

Representation issues of logic-level communication include the latching of input values into registers, sensing and assertion of signal levels, and the output of values calculated in the data path. Interacting digital systems may exchange both data values and control signals. Regardless of the nature of the exchange, it must

conform to a predetermined *protocol.* Data transfer may be effected using a handshake procedure, possibly along with a packet formation procedure. Some control signal protocols are very simple: a reset signal may directly impact the functioning of a circuit, for example. Others, such as interrupt handling, are more complex.

Behavioral specifications for interfacing processors therefore describe data and control exchange along with protocol signal events. The specification describes the occurrence of logic levels on signal lines, including values, sequencing, and relative or absolute time constraints between events. Signals can range in size from single to multiple bit.

The distinction between data and control signals is based primarily on use. Values that are read in and manipulated by data operations to produce new values are data values. Conversely, values read in and used to determine the conditional branching behavior of the system are classified as control signals. For those values that have a dual control/data character, both representations can be used in parallel without difficulty. The value is read in: its effects on system branching behavior are reflected in the timing graph, and the data manipulation performed with it is shown in the data flow graph. Modifications to the value arising from data path operations, as defined by the contents of a particular register for example, will not adversely affect branching behavior. This is because the synthesis method never assumes the persistence of a signal beyond a single clock period, and because the control clock period is never longer than the data path clock.

In the case of output signals, those that are produced by data manipulating operations are data values. Output signals defined as bit constants in the specification are control or protocol signals. [3.2] This distinction is not an absolute, and reflects implementation considerations rather than an immutable dichotomy, as will be seen later.

[3.2] It is generally more efficient to consider large constants as data values to be stored in registers. This is an implementation decision, however.

3.1.3 Control Flow

In addition to the control constructs normally found in structured programming languages, such as loops and conditional branches, the following control characteristics are of special importance for interfacing processors:

1. Sequencing constraints impose a partial order on events, indicating which events must occur in strict sequence and that may occur in parallel. The sequencing requirements of interfacing processors cannot be determined by examining data dependencies alone, unlike traditional data path synthesis specifications. This is due to the interaction with the outside world, and the control nature of many of the signals.

2. Parallel branches must be supported, particularly since the specification may cover the dual tasks of internal processing and external communication.

3. Process initiation can be based on the assertion of a particular signal or combination of signals, rather than on calls executed by other processes. Whenever the initiation signal is detected, the process should be started.

4. Priority arbitration may be imposed on a group of processes or events. The initiation of events and processes then depends not only on initiation conditions, but on the defined priorities. This is true when only one of the events or processes of the group should be performed at a time. If more than one initiation condition is true, the process with the highest priority is initiated.

5. Interrupts may occur at any time; interrupt signals may be synchronous or asynchronous external signals. When an interrupt occurs, current processing ends and the interrupt routine takes over. If an interrupt is disabled, the current process can continue to its end before the interrupt routine takes over.

3.1.4 Timing Behavior

Hard real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time

at which results are produced [SR88]. Real-time constraints are an aspect of most systems that interact significantly with their environment. Interacting systems are constrained by communication requirements to observe both relative and absolute timing specifications. This design requirement is an important distinction between traditional data path synthesis and interfacing processor synthesis. We will consider three general categories of timing behavior of special relevance to interfacing processors:

1. Time constraints specify a limit, either minimum or maximum, on the duration of some combination of events, or on the time between two distinct events of the interface behavior. Time constraints are used to indicate when events must occur simultaneously.

2. Time intervals exist in the form of delays or holds on the assertion of signals for an interval that is either bound or random. It may also be necessary to measure the time of intervals bounded by some events or to perform certain operations repeatedly for a specified period. In these cases, the required measurement accuracy or periodicity is specified.

3. Synchronization is a precisely defined relationship of the timing of events with each other. This is an important function performed by systems that interface with their environment. Communication between independent processes may be synchronized in one of two ways. First, certain events may be constrained to occur in synchronization with a predefined, periodic signal called the *clock*. Synchronous signals arise from either (1) the processor's own data path or (2) an external device that shares the same clock with the device being designed or whose clock has a known rate. Second, events may be synchronized to other signals besides the clock. In the case of asynchronous communication protocols, the communicating devices do not share a common clock. Nevertheless, some of the protocol signals act as synchronizing agents for the receiving processor. For example, when an asynchronous *data ready* signal is detected by the receiving device, it can be assumed that the data, that is synchronized to the *ready* signal, is available.

New Construct	Function	Original	Function
priority-chain	priority encode/decode	table; if	look-up; cond.
time(lbls,rel,t)	time constraints	none	
syncr, syncf	synchr to rise, fall clock edge	sync	synchr to clock
seq	patt. match, non-sliding wind	delay until	sliding window
while(t)(c)do	do while (c), time out after (t)	while	no time out

Table 3.1: Extensions to SLIDE

3.2 Specification Language

The SLIDE Hardware Descriptive Language [PW81] was designed for the description of input/output, interfaces, and interconnected digital systems. Its development was based on a behavioral analysis and classification of such systems done by A. Parker [Par75]. The syntax and semantics of the language are similar to other procedural languages, both programming and hardware descriptive, and particularly to the ISPS hardware descriptive language. Here we briefly summarize the features of SLIDE that any HDL must have if it is to be used as a specification language for interfacing processors. Table 3.1 summarizes some extensions made to the language; all extensions are discussed in the text, where they are explicitely denoted by an asterisk (*) to distinguish them from existing constructs.

3.2.1 Data Manipulation

SLIDE provides all the standard operations of programming languages such as addition, multiplication, division, subtraction, and logic functions. In addition there are some operations especially relevant to interfacing processors. SLIDE has a special declaration for associative memory tables. Tables are specified using the keyword *Table*, a name, and the bit-widths of the table input and output. This is followed by a list enumerating the input and corresponding output values. Tables are accessed using the operators *encode* and *decode*. Encode accepts an input value and returns the corresponding output. Decode accepts an output value and returns the corresponding input.

Bit operations in SLIDE include the specification of bit slices and concatenating operations. Even and odd parity operations are also included. Bitwise logical

operators are *not, or, xor, and* and *eqv.* The *format* operator makes an arbitrary bit pattern from two sources.

3.2.2 Logic Level Communication

In behavioral specifications, internal hardware is not described. It is necessary, however, to explicitly define hardware on the environmental interface. This hardware is taken as a given by the synthesis system, and cannot be modified. Interface hardware that can be described in SLIDE includes registers, memory arrays, lines, and buffers.

Lines can be defined as synchronous or asynchronous (this will be elaborated on in the next section on control flow). Signal transitions on lines, from low-to-high or high-to-low can be described, as well as low or high logic levels. Logic may be further defined by specifying a number of technology terms for interface lines, including *tristate* and *open collector active low.* Technology terms are user definable. Binary and octal values may be defined. Writes to external lines are represented as assignment statements.

3.2.3 Control Flow

3.2.3.1 Sequencing and Branching

There are two sequencing constructs in SLIDE. The delimiter *next* is used to indicate that the statements preceding are to be completed before the statements following the *next* are executed. Parallel branches are represented by separating statements by a semicolon, indicating that they must be executed in parallel with each other. Statement groups are formed through the use of parentheses. The delimiter, either *next* or a semicolon, placed after a statement group applies to all constituent statements.

Conditional execution is represented in SLIDE by the *if...then...else* statement. Loops are represented by *while, until,* or *loop* statements.

3.2.3.2 Process Initiation

The central execution unit of the SLIDE language is the process. Processes begin execution whenever their initiation conditions become true, rather than being called procedurally by another process.[3.3] This is represented by the reserve word INIT:

```
INIT Iarbiter:0 WHEN initline EQL 1;
INIT Dtransfer:1 WHEN dataready EQL 1;
```

These two statements indicate that the process named Iarbiter begins execution if and when *initline* is asserted to be logic level 1, whereas Dtransfer is initiated when *dataready* is asserted.

3.2.3.3 Priority

Process priority is indicated in SLIDE by a number attached to the process name by a colon, lower numbers indicating higher priorities. In the example above, Iarbiter, with a priority of 0, has higher priority than Dtransfer, with a priority of 1. Process priority is necessary whenever only one process can be executing concurrently. Whenever more than one process initiation condition is true, the one with the highest priority is initiated.

Priorities associated with events other than processes can be specified in SLIDE through use of the *Table* declaration, along with *encode* and *decode* operations. In the case of bus grant requests, for example, only the highest priority bus request will be granted by an arbiter. For three bus request levels, the table would look like the following, where values on the left represent bus request signals, and values on the right represent bus grant signals:

```
Table bus-request < 2 : 0 >< 2 : 0 >
    '000 ⇒ '000,
    '001 ⇒ '001,
    '010 ⇒ '010,
    '011 ⇒ '010,
```

[3.3] Subroutines can also be defined.

'100 ⇒ '100,
'101 ⇒ '100,
'110 ⇒ '100,
'111 ⇒ '100,

The difficulty with this approach is that the relationship between the table and the bus request and grant lines is not obvious. The input to the table would be a variable defined as a concatenation of the individual bus request lines. The output would be similarly defined as a concatenation of the individual bus grant lines. Another problem is that it would be difficult for a synthesis system to detect from this representation that library modules designed for priority encoding and decoding could be used. This is because the *table* construct is more general than the priority function.

Another method to represent priorities in SLIDE uses nested *If* <*exp*> *Then* <*stmt*><*else-clause*> statements. Each succeeding <*exp*> checks for the next lower level priority. The associated <*stmnt*> is executed only if no higher priority conditions are true. This is less concise than the table method, and suffers from the same obscurity for implementation purposes. The timing of the sequence of *if* statements also is underspecified and unclear. Should the conditional checks be performed one after the other, or simultaneously? Because the writer of the specification is aware of the priority scheme, it makes sense to provide a SLIDE construct to explicitly preserve this information. The form of the proposed construct resembles that of a *case* statement, as follows:

```
PRIORITY-CHAIN*
begin
Condition1: Actions1
Condition2: Actions2
Condition3: Actions3
Condition4: Actions4
        ...
end
```

Conditions are listed in order of their priority, highest priority first. All conditions are tested in parallel, and the actions of the highest level condition that is true are executed. The conditions may be Boolean variables that are true or false, such as interrupt request lines that are asserted or not. Conditions may also be more complex, as shown in the example below from the UNIBUS protocol where priority levels can be set to ignore low priority requests. The "slashes" in the specification represent rising and falling edges of the clock.

```
PRIORITY-CHAIN*
npr:
        begin
        npg_/; delay 5000 until sack_\; npg_\
        end
(br7 and pri lss 7 and ready):
        begin
        bg1_/; delay 5000 until sack_\; bg1_\
        end
(br6 and pri lss 6 and ready):
        bg2_/; delay 5000 until sack_\; bg2_\
        end
        ...
```

In this example, the condition *npr* has the highest level priority. Whenever it is asserted the bus is granted to the requesting device. The other bus request lines br7, br6, etc., are only acknowledged if the priority is set to the proper level, as measured by the variable *pri*, and if *ready* is asserted. The corresponding action in each case is to assert the corresponding bus grant signal, delay until an acknowledge is received (timing out if not received within 5000 ns.), and then deasserting the bus grant signal.

The synthesis system is informed through use of this operation of the existence of a priority function, and can decide on an implementation accordingly. It may chose to use predefined priority modules or not, according to the particular problem characteristics.

3.2.3.4 Interrupts

Interruption is the *premature termination* of a sequence of events brought about by some event external to the sequence. The notion of interrupts in SLIDE is linked with the definition of the process. Permission to terminate a process is given only to those processes with a higher *priority* than the executing process. An executing process may *disable* interrupts, and thereby guarantee normal termination and enforcement of *critical sections.* Interrupts are disabled by setting a 1-bit variable defined for that purpose. Whenever the initiation condition of a process becomes true, that process may begin execution providing that (1) no process with the same or higher priority is executing or has its initiation condition true, and (2) if a lower priority process is executing then interrupts are not disabled. Any lower priority process currently executing is then terminated. The granularity of the interrupt action is established by the processes that are defined. CPU interrupts, for example, are accepted at precisely defined steps of system operation. A SLIDE description of the CPU would therefore not associate the interrupt with the highest level process, but with a low level process corresponding to that phase at which interrupts can be accepted. Alternatively, a synchronous branch governed by the interrupt signal can be defined within the description.

3.2.4 Timing Behavior

3.2.4.1 Time Constraints*

SLIDE does not have time constraints, primarily because it was designed to be descriptive, rather than prescriptive. Time constraints, unlike time intervals, primarily refer to performance requirements on implementing hardware. To use SLIDE as a specification language for synthesis, therefore, we must introduce minimum and maximum time constraints.

There are two possible approaches to this problem. The first, similar to that proposed by Girczyc [GK84], is to associate constraints with ranges or blocks of code. The second, proposed by Nestor [NT86], is to attach labels to individual statements in the specification and separately specify time constraints between pairs of labels. The second approach is more general and powerful, in that any

conceivable constraint can be expressed.[3.4] For this reason, we will use this approach. Following the convention proposed in [NT86], labels for statements are placed immediately before the statement delimiter and constraints specified as follows:

<Statement 1> (L: label1) NEXT
<Statement 2> (L: label2)
TIME(label1, label2) {GEQ,LEQ,LT,GT,EQ} time

The constraint indicates the time limit between the beginning of execution of the first and the beginning of execution of the second labelled statements. The label method is perhaps too powerful in its unrestricted generality, since constraints can be expressed between arbitrary branches of conditional code, or other ambiguous locations. We will rely on the user to follow rules on the placement of constraints. In particular, constraints cannot be imposed across conditional or parallel branches. In a deviation from Nestor's approach, a constraint will be allowed on a single statement using a single label, indicating a time limit for the specified operation. A time constraint imposed on a single compound statement like a *do* loop, or on a statement group formed using parentheses, constrains the operations within as a whole. This convention follows the more structured approach taken by Girczyc.

3.2.4.2 Time Intervals

Time intervals are represented in SLIDE by the delay statement: *delay* <*time*>. To assert outputs for a specified time interval, the output statement(s) is placed in parallel with the *delay* statement. It is sometimes desirable to perform some action repeatedly for a certain time period, for example, to count the number of times a particular event occurs. The *delay* statement is inadequate for this purpose.

One way of representing time limited repetitious behavior in SLIDE is to define a loop enclosing the behavior. To achieve the proper duration, the number of loop iterations must be adjusted in conjunction with a time constraint on the enclosed behavior. Although this sounds tricky, in fact such detail must be a part of the

[3.4] With the possible exception of a constraint on the execution time of a single statement.

specification. For example, if the events being counted are synchronous, the clock period must be known. If they are asynchronous, it must be known how close together the events can be so that the implementation can catch all possible events.

A conceptually more direct representation, however, can be achieved by extending SLIDE to permit use of a time constraint as a loop exit condition. The time constraint on the loop is identical to the time period during which the behavior should be repeated. Additional information will generally be needed to specify the frequency or speed with which the constituent operations should be performed. This time will equal the duration of each loop iteration. The way in which this is specified will depend on the behavior. Behavior related to synchronous signals will be defined by the specified clock period of the signals themselves.[3.5] Asynchronous behavior or data manipulating operations performed repeatedly will require associated time constraints for full specification. The proposed *while* statement incorporating a time constraint as exit clause would look as follows:

```
WHILE <time> DO <stmnts>
```

To *measure* a time interval that is bounded by two events, a count is initiated by the first event. A delay equal to the units of time to be measured is place in parallel with the count operation. As an example, the time between the arrival of two asynchronous inputs, *start* and *finish*, is to be counted in seconds. The following specification can be used:

```
DELAY UNTIL start EQL 1 NEXT
WHILE finish EQL 0 DO
BEGIN
delay 1 (sec); Count = Count + 1
END
NEXT
```

Processing is delayed until *start* is asserted. The *while* loop is then entered, and is not exited until *finish* is asserted. At each loop iteration, *count* is incremented and a delay of one second is produced. Therefore, once a second, between the

[3.5] The representation of synchronous behavior in SLIDE is further explored below.

assertion of *start* and *finish*, *count* is incremented, thus measuring the time interval in seconds bounded by the two events.

3.2.4.3 Synchronization

Events may be synchronized by an explicit and consistent relation to a regular, repetitive signal called the *clock*. Signals might change only on the rising edge of the clock cycle, for example. Events and signals of this kind are termed *synchronous*. Input/output synchrony is represented in SLIDE using the keyword *sync*, that is used to define synchronous signals. To accurately represent specifications that need a more detailed description, it is proposed to extend SLIDE by the addition of two new keywords, *syncf** and *syncr** meaning signals that change on the falling and rising edges of the clock, respectively. The use of a synchronous variable implies an implicit relation between the clock signal and the testing of the variable's value. Lines specified as synchronous can be tested for a *time-ordered* sequence of values on the line, using the *delay* statement:

```
SYNC @ 5000 LINE x<>
DELAY UNTIL x EQL |0|1|1
```

The first statement identifies x as a one bit synchronous signal with a clock period of 5000ns. The second statement indicates that execution is to be delayed until x exhibits the time-ordered sequence of values *0,1,1*.

The SLIDE *delay* statement specifies that the value of x is to be tested using a sliding window, that is, the pattern should be detected wherever it appears in the input stream. Another pattern matching strategy that cannot be represented using the *delay* statement uses a grouping approach; the bit stream is treated as n-bit words, where n is the number of bits in the pattern, each of which is examined for a match. Specifications may require either or both behaviors, with very different results.

An additional inadequacy of the *delay* statement is that multiple patterns may be sought using the grouping approach. For example, if the three bits have the value *0,1,0* then A is incremented, else if the three bits have the value *0,1,1* then B is incremented, else no action is taken. The use of parallel *delay* statements does not capture the proper behavior.

Furthermore, the search for patterns in an incoming bit stream may not be a one-time action, but a continuous behavior. It is then necessary to represent (1) the length of time the check is to be made, and (2) actions to be performed in case the pattern is detected, including when the next check should be performed. Following is an example specification from [Com84]:

> A serial data line X is allowed to change on the falling edge of the clock signal. If the sequence 10 occurs on the data line, output Y should be asserted for 1 clock period before the next check is started. If 01 occurs, output W should be asserted for one clock period before the next check is started. If 00 or 11 appear as the two sequential bits on X, no output should be generated and the next check should begin after a delay of one state time.

To represent a non-sliding window check, it is proposed to introduce a new operator, *seq* <line> <bit-pattern><stmnts>*. Because multiple bit patterns may be sought, the statement resembles a *case* statement, with the difference that the match values of the case statement are replaced by patterns representing sequential bit values. To represent the behavior described in the above paragraph we would write:

```
WHILE TRUE DO[3.6]
BEGIN
SEQ* <Z>
10:  NEXT Y←1 NEXT Y←0;
01:  NEXT W←1 NEXT W←0;
00:  NEXT NOOP NEXT NOOP
11:  NEXT NOOP NEXT NOOP
END
```

The bit value being examined is specified by <Z>. Corresponding actions are represented alongside each pattern, following the colon. The delimiter *next* is used to specify sequential execution, that is identical to the next clock period for synchronous specifications. The semicolon delimiter is used to specify parallel

[3.6]This is SLIDE's infinite loop construct.

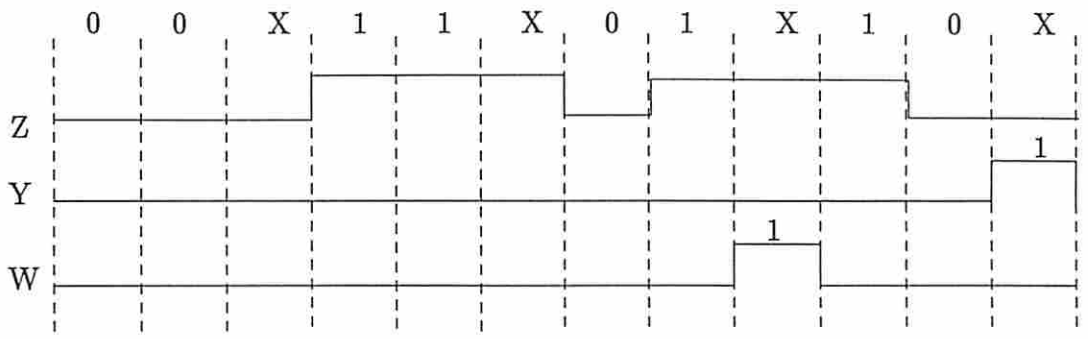


Figure 3.1: Timing Graph Representation

execution. A timing diagram is shown in Figure 3.1 to clarify the representation for the specification above. The values of Z are shown above the timing graph, using X to indicate don't-care for the value during the third clock period. If either sequence 00 or 11 occurs, no outputs are asserted. If Z exhibits the sequence of values 10 then Y is asserted in the following clock period. Y is deasserted in the next clock period; this deassertion is performed in parallel with the next statement, as indicated by the semicolon. The next statement is the next iteration of the loop. This indicates that bit stream is to be examined every three bits rather than every two: one bit is *skipped*, regardless of the pattern read in. The deassertion of Y is performed in parallel with the read of the first bit of the next group. If no bits should be skipped, then the two steps for each case should be enclosed in parenthesis. For case 10 this would read: 10: NEXT (Y←1 NEXT Y←0); indicating that the operations should be performed in parallel with the next two reads of Z.

To test for a bit pattern in a sliding window, the existing *delay* statement can be used. Each bit value coming in is compared to the first bit of the bit-pattern until a match is found. An example specification is:

> When the sequence *1,0,1,1* is detected on line X, the record count is to be incremented and the check for the next sequence begun immediately.

A SLIDE representation of this behavior is as follows:

```
WHILE true DO
BEGIN
DELAY UNTIL X EQL |1|0|1 NEXT
```

```
IF X=1 THEN Record-Count = Record-Count + 1 NEXT
END
```

When the pattern is detected the IF statement is executed during the next time step. If the value of the next bit is one, Record-Count is incremented during the same time period (since there is no delimiter NEXT after THEN). Otherwise the loop is reentered and the delay statement executed at the next time step. In this way no bits are missed.

Events may be synchronized to some signal other than the clock. This is particularly necessary for asynchronous communication between independent processes. The detection of a *data.ready* signal is used to synchronize the exchange of data on bus lines, for example. In SLIDE, this type of synchronization is represented by a statement delaying execution until the synchronizing signal is detected. The statement is of the form *delay <exit clause>* statement, in which *exit clause* includes the keyword *until* or *while*, and indicates a value for the asynchronous input. Because synchronization between independent processes may fail, it is desirable for the receiving process to detect and recover from it. The specification may include a time-out clause for this purpose. Timeouts are represented in SLIDE by appending a time to the *delay* statement, that becomes *delay <time><exit clause><else clause>*. The optional *else clause* indicates an action to take in case the time elapses before the exit clause becomes true. To specify that outputs should be asserted during the delay, output statements are placed in parallel with the delay statement, using the semicolon.

It may be desired to repeat a particular behavior during the wait for synchronization. In this case, the *while <cond> do* statement can be used, referring to the asynchronous input in the <cond>. The *while* statement does not have a time-out clause, since it was intended for synchronous operation. It is proposed to introduce such an optional clause, giving a form similar to the *delay* statement:

```
WHILE* <time><cond> DO <statements> ELSE <else clause>
```

3.3 Internal Representation

3.3.1 Controller Versus Data Path Implementation

The Design Data Structure (DDS) was developed at USC by J. Granacki, D. Knapp, and Parker [Gra86, KP85].[3.7] There are four *models* in the DDS for *Data Flow, Control and Timing, structural*, and *physical* information. Behavioral specifications make use primarily of the Data Flow and Control and Timing Models; register-transfer level information is derived during high-level synthesis and stored in the structural model; the final implementation of the design at the layout level creates the information in the physical model. Library modules are stored in the physical model, however the behavior of the modules is defined in the Data Flow and the timing Models. The relations between operations, timing, structure, and geometry are explicitly represented using *bindings* between related items in the different models.

The DDS represents behavior using the Data Flow and the Control and Timing Models. By the definition of the algorithmic state machine technique, recall that the separation of behavior into data path and control is not unique: many partitions are possible. If the behavior represented in the Data Flow Model is realized using data path synthesis techniques, and the behavior in the Control and Timing Model is implemented by a control synthesizer, then the form of the DDS implies a particular partition of the underlying implementation.

The way in which a SLIDE description is translated to the DDS can therefore impact the form of the implementation. For every SLIDE specification, there may be many DDS representations that will capture the same behavior, with possible differences resulting in the implementation. To account for this multiplicity, we can either (1) define multiple translations to DDS for each SLIDE construct, or (2) define a single "canonical" method of translation, and consider alternative partitions later, during synthesis itself. Although control/data path tradeoffs are

[3.7] An overview of the DDS is given in our paper [HPG88] (Appendix A). Granacki shows a lifetime representation for each data flow value. The method used here omits the lifetime interval and associated time constraint arcs, since the lifetime is established sufficiently by the use of the value as data or control in the specification. Also, the time constraint arc is no longer considered in the arc count associated with a point of the timing graph. This results in far fewer *and* branch and *and-join* points.

not addressed in this thesis, the representation and synthesis techniques developed for the controller have been done with the later technique in mind for future development.

The following sections show how the behavioral categories of interfacing processors described in section 3.1 will be represented in the DDS, and how the SLIDE constructs described in section 3.2 relate to the DDS representation.

3.3.2 Data Manipulation

The Data Flow (DF) Model is a bipartite graph with two node types, *value* and *operation*. Each value and operation of a particular data flow graph is defined as an instance of an abstract *type* that defines its characteristics and behavior. An operation is primitive if there is an operator available in the module library that can implement that operation. The number and types of value and operation primitives is therefore not fixed in DDS. Arithmetic operations such as *add* and *subtract*, as well as boolean operations such as *greater than*, are represented in the Data Flow Model of the DDS as primitive data flow operations. The specification of bit slices of a value is represented using another primitive DDS data flow operation *split*, that has inputs defining the number and location (offset from right) of bits to be split off. Concatenation of bits to produce a new value is represented by the operation *concat*. Bit-wise operations such as *or, xor* and *eqv* are also defined as primitive operations. More complex operations, such as priority encoding, can also be defined primitives if there is a library operator, such as a priority encoder, to implement them.

Data flow operations can be defined hierarchically. Using a hierarchical definition, the specification may employ a high-level operation, such as a priority encode function, even when there is no comparable operator in the library. The type definition of the operation defines the way in which it is composed of lower level operations. The *format* statement of SLIDE can be defined as a hierarchical operation, its action defined by its decomposition into bit operation primitives.

Table and memory accesses can be represented in DDS as operations that are bound before data path synthesis to the proper modules. The *decode* or *read* operations have a single input, the address, and a single output, the data. The

encode operation has a single input, the data, and a single output, the address. The *write* operation has no outputs and two inputs, the address and the data. Each access to the table or memory is specified by a separate instance of the appropriate operation. The contents and behavior of tables, but not that of memory, are specified in the operation type definition. *Table* operations as defined in SLIDE may be translated to DDS in this way.

Sequential modules and those whose performance relies on clock edges cannot be represented using the simple delay model that is adequate for combinational components like *and* gates or adders. This is because the delay model assumes that available design primitives can be placed into time slots of any length, cascaded one after the other, or scheduled into multiple time slots, provided that the total delay from inputs to outputs is accounted for. An additional assumption is that modules may be shared among operations of the same type provide that the operations are not in the same time step. This is only true for combinatorial operators; the assumption is violated for sequential operators.

The use of sequential components reflects an implementation decision: a behavioral specification will not stipulate their use. For this reason SLIDE does not have special constructs for the representation of shift registers, counters, etc. One of the tasks of synthesis is detection of those behaviors that can and should be implemented using such components.

DDS can be used to represent physical components, since it is able to represent the implementation as well as specification. Modules that have state information, such as counters and shift registers, must have their relations to the clock period represented. In the DDS, every physical component is represented by a structure and also by its behavior, behavior being defined as data flow plus control and timing subgraphs with relationships between the two represented as bindings. A sequential component's data flow graph definition can serve to identify its function, and its timing graph to describe its control requirements and characteristics.

A synthesis system could recognize the possibility of using a sequential component by examination of the behavior described in the specification. During synthesis the component, unlike combinatorial logic, cannot be shared arbitrarily between operations in the data flow graph. Its functioning cannot be cascaded from one clock period to another nor placed in series with other operators in the

same clock period. The module binder must also be aware that sharing of such components between operations is restricted, and must also be able to determine when the proper control signals, such as *load* or *initialize* should be asserted. Because this is largely a data path synthesis issue, and this thesis focuses on control synthesis, the use of sequential components will not be pursued further.

3.3.3 Logic-Level Communication

Interface behavior descriptions include references to *physical lines* and *registers* that may be read from or written to. Their representation in the DDS has three aspects: (1) the value on the line or register is represented by a value in the Data Flow Model, (2) the hardware is represented in the Structural Model, and (3) timing characteristics related to interface hardware are described in the Control and Timing Model.

The behavioral implications of technology terms such as *tristate* and *open-collector-active-low* are of greatest importance for lower level synthesis. Tri-state lines, for example, require assertion of a control signal in addition to the value itself to permit the line to be driven. The DDS representation of such technology terms makes use of type definitions for values and carriers. If voltage levels are implied by the values *0* and *1*, then the specification needs no further explanation for high-level synthesis purposes. If false and true logic levels are used instead, then the module binder and system that determines the values of control signals must be aware of the technology type.

Data flow values are associated with defined *types*, that indicate bit-width. Type definitions for values may include field definitions. For example, complex numbers can be defined with real and imaginary bit fields. Constants may also be represented as special types in the data flow graph. Lines and registers are referred to as *carriers* in the Structural Model; different types of carriers may be defined, specifying bit width and other characteristics. The bit width of the carrier must be adequate to hold the specified value.

The *writing* of values onto interface lines is represented in DDS using bindings between a signal carrier defined in the Structural Model, a value defined in the Data Flow Model, and a time interval in the Control and Timing Model. Bindings are

designated by *Binding(carrier,value,interval)*, indicating that the specified value must be asserted on that carrier during the identified interval (see Figure 3.2). Several different data flow values may be bound to the same carrier during non-overlapping intervals.

Values in *output* bindings are either constants, calculated by operations of the data flow graph, or retrieved from tables or memory. Constant, binary values of various bit widths, such as *0* and *1*, are predefined in the Data Flow Model. These constants are bound to the proper interface line(s) during the designated interval. Signal transitions, as opposed to levels, may be defined in the Data Flow Model as well. A value of type *low-to-high* or *high-to-low* transition can be included in the Data Flow Model and referred to in the binding.

The representation of the *reading* of interface lines or registers has two aspects. First, the time interval during which the input is valid is specified; second, the function of the value being input is described. Input timing may be defined using a binding similar in form to the output bindings. *Synchronous* bindings indicate that the input is valid during the entire interval to which the value is bound. Its form is *Binding(interval, carrier, value)*. *Asynchronous* bindings indicate that the input may be asserted at some point during the interval to which the value is bound. The asynchronous binding, {*Binding(interval, carrier, value), point*} is quite different in form and effect from the synchronous one. Its use and function will be described in detail in the next section.

Systems that interact asynchronously may exchange data values, represented as global variables in specifications. Asynchronous communication is accomplished through use of *handshaking protocols* of various types and complexity. Such protocols are designed to guarantee the correct reading of data being exchanged, since transitions on multiple lines may be skewed. Handshaking protocols use bit signals, such as *data ready*, that are received asynchronously and indicate that the value on the data (or address) lines is valid and can be read. Although values being read from interface lines are not synchronized to a known clock, they *are* synchronized to the received communication protocol signals. For this reason, these values are represented using synchronous bindings; whereas the behavior of handshaking or protocol signals is represented by asynchronous bindings.

If the value read from interface lines is to be latched into a local register, this is represented in the DDS by specifying a binding between the data flow value and the local register during the proper time interval.

The value derived from a read of interface lines performs some function in the specification. This may be either a data or a control function. The representation of the behavior of a value read from interface lines is very similar to the representation of internal values. Data values and their behavior as inputs to data operations are represented in the Data Flow Model. The DDS uses a single assignment data flow structure, which means that each value is only used once. If values on interface lines are accessed more than once in a specification, then it is necessary to associate the corresponding data flow value with a loop index. The index guarantees a unique name for each value. The time interval during which each indexed value is alive is defined by loop constructs in the timing model. If the signal in the specification is synchronized with respect to a predefined clock, its index is updated every clock cycle. The indexed values are also bound to a structural carrier corresponding to the interface line(s). An example is an address register, *addrreg*, which the system must check for its own address value. The value contained in *addrreg* is represented in the Data Flow Model as an indexed value bound to a structural carrier named *addrreg*. Values read from global lines that are defined as variables in SLIDE specifications are translated to DDS in this way.

If the value read on a global line is used as control, the DDS representation of the signal includes a value instance in the data flow, which defines its *type*, and a description of its control behavior in the Control and Timing Model. The behavior of interface protocols as well as the effects of control signals arising from external sources are largely represented in this way. Figures 3.14 and 3.20 in the next section illustrate this representation.

3.3.4 Control Flow

Control flow information includes the proper sequencing of and conditions causing events. This information is represented in the Control and Timing (CT) Model of the DDS. The CT model is a directed graph whose points represent events, such as the initiation or termination of an operation, and whose arcs represent

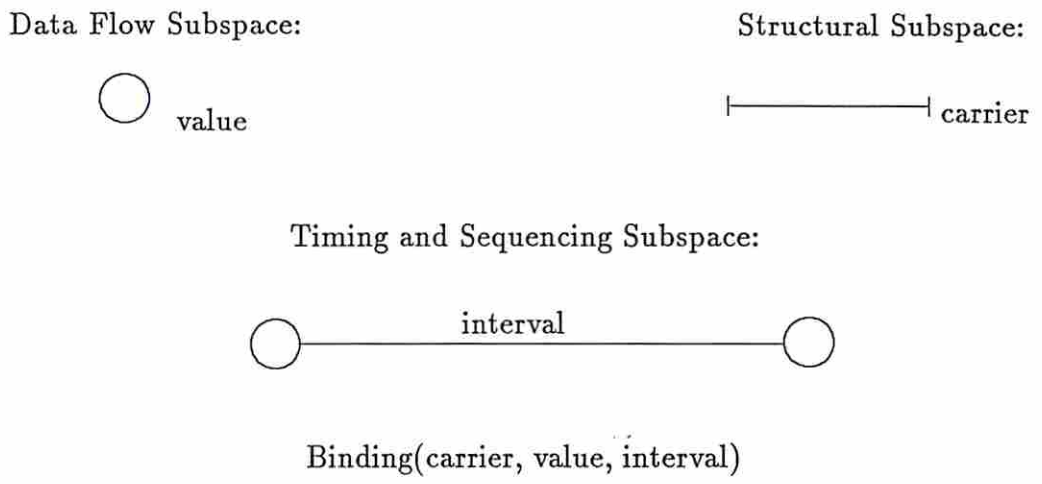


Figure 3.2: Representation of Outputs in the DDS

relations between events, such as operation durations and constraints. There are four basic arc types: time constraint, time interval, causal, and glue arcs. The glue arc serves only to connect points of the graph and does not reflect any behavior. The significance of the other arc types will be explained in subsequent sections.

The nature of the control flow from one interval to another is specified by the type of the connecting point. There are seven point types: simple, and, and-join, or, or-join, begin-loop, and end-loop. The semantics of the different point types will be described below in connection with the different behaviors they represent. Each point and arc of the graph must be given a unique name. To simplify specifications, names can be derived from an abbreviation of the arc or point type, appended to a unique number. An interval arc might be named *i20* and a begin-loop point *b2*, for example. Simple points are labelled with a *p*. The number may be omitted in figures (though not in specifications) when unnecessary for the explanation.

Associated with the CT graph are *bindings* specifying the connections between particular time intervals of the CT graph and items described in the data flow or structural models. The schedule chosen for data path operations, for example, will be indicated by binding operations in the data flow to the correct interval sequences in the CT model.

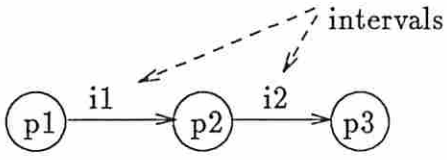
3.3.4.1 Sequencing

Sequencing requirements that are not related to data dependency are explicitly represented in the CT graph. Simple event precedence is represented in the DDS Timing Model by binding the sequential operations or events to consecutive time intervals separated by simple points. Simple points indicate time order only: events associated with the ingoing arc precede those of the outgoing arc. Operation precedence based on data dependency is depicted in the data flow model, which reveals the maximum possible parallelism between operations. This situation is illustrated in Figure 3.3a. The use of interval arcs separated by a simple point, with operations bound to the intervals, corresponds to SLIDE statements separated by the delimiter *next*.

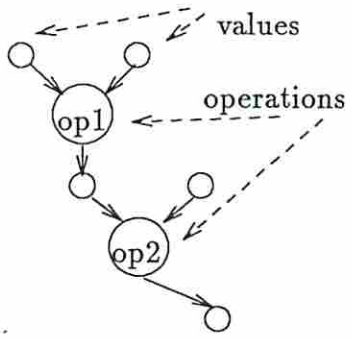
Statements or processes that execute at the same time or in parallel are concurrent. Concurrency is represented in the DDS by an *and* node in the Control and Timing Model with a single in-coming arc and one out-going arc for each of the parallel statements or processes, as shown in Figure 3.3c. There may be a sequence of arcs on each branch. In that case, there is no explicit time relationship between different points on parallel branches. Referring to the same figure, the events bound to interval *i2* do not *necessarily* occur after the events bound to interval *i3*, for example. If the branches come together again, they must do so at an *and-join* point. Any events bound to intervals succeeding an and-join must wait until all events on converging branches have terminated. The parallel execution of statements represented by the *and* node of the DDS is identical to the semicolon delimiter in SLIDE. The and-join point corresponds to a *next* statement in SLIDE that follows a group of parallel statements.

During synthesis, the order of operation execution may be modified in two ways from that described in the specification. Concurrent operations may be executed in series, or sequential operations may be placed together in one long time interval or clock period. Simultaneous operations, indicating operations that are scheduled into a single time slot, are represented in the DDS by binding operations to the same time interval in the Control and Timing Model (Figure 3.3b).

Timing and Sequencing Subspace

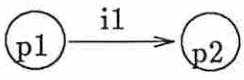


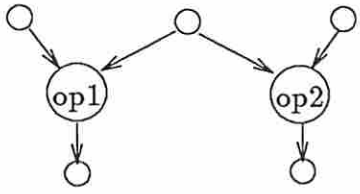
Data Flow Subspace



a. Sequential Events

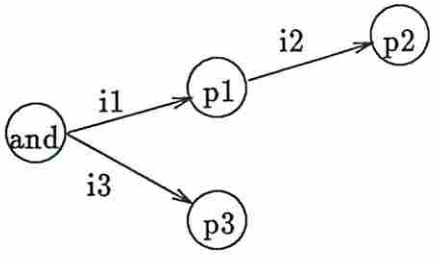
Binding(op2, i2)
Binding(op1, i1)

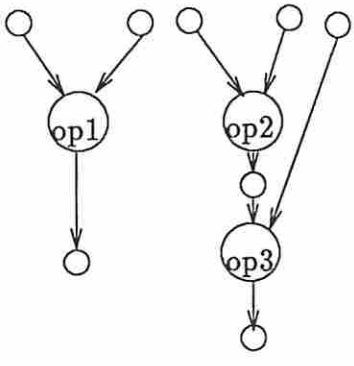




b. Simultaneous Events

Binding(op1, i1)
Binding(op2, i1)





c. Concurrent Events

Binding(op1, i3)
Binding(op2, i1)
Binding(op3, i2)

Figure 3.3: Event ordering

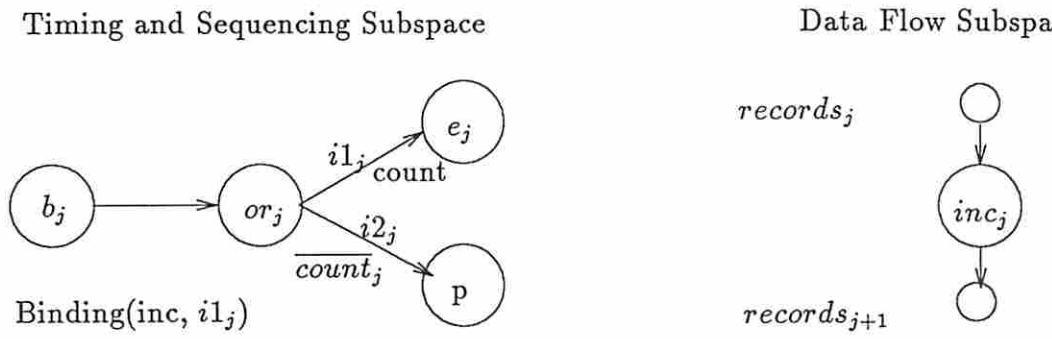


Figure 3.4: While Statement

3.3.4.2 Conditional Branching

Conditional branches are also represented in the CT Model. Two types of values may be involved in determination of conditional branching behavior: synchronous and asynchronous. Synchronous values arise from the data path or are inputs from the environment that are synchronized to a known clock or some other signal. The DDS representation of a synchronous conditional branch employs *or* points. Each mutually exclusive branch of an *or* point is associated with a *synchronous predicate*, which is a value or Boolean function of values defined in the data flow graph. The predicate of only one arc can be true at a time, indicating which branch is taken. The branches of an *or* point can terminate at *or-join* points. This representation of the conditional branch corresponds to SLIDE's use of the structured programming statement *if...then...else.*

An example of a conditional branch is shown in Figure 3.4, which represents a while statement. For now we ignore the outer loop behavior, which is explained in the next section. The conditional branch occurs at the *or* point. The value *count*, a synchronous input, acts as a predicate that chooses between two intervals: *i1* which has the operation *inc* bound to it, and *i2* during which no action is taken.

Asynchronous inputs are values read from the outside that have indeterminate timing with respect to other system events and operations. Although an exact time is not known, the range of time during which the asynchronous input may or will occur must be specified. For the most extreme case, this range covers the entire specified behavior. This is true of a *reset* signal, for example. We will make use of the asynchronous predicate in DDS to represent the behavior associated with

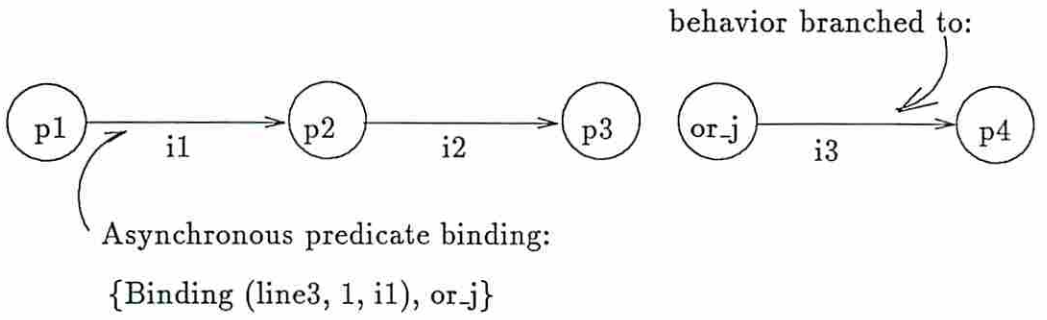


Figure 3.5: Asynchronous Predicates

asynchronous input signals. Asynchronous predicates are associated with intervals rather than points in the CT graph. The underlying semantic model assumes an infinite number of conditional branches along the interval to which the predicate is bound [Gra86]. When the asynchronous predicate becomes true at any of these points along the interval, a branch is made to a specified point. The consequent behavior is described by the graph following the branch point. This is represented in the DDS by an *asynchronous predicate binding*, of the form

{Binding(interval, signal, value), point}

This specifies that if *value* is detected on the structural carrier *asynchronous-signal* at any time during *interval* a branch is made to the CT subgraph that defines subsequent behavior, beginning at *point.* To represent the effects of an asynchronous input in CONSPEC specifications, the name of the input will be assigned to *signal.*

An example is shown in Figure 3.5. The asynchronous binding is attached to interval *i1*, and *or_j* is the point to which branching occurs. Interval *i3* has bound to it those operations performed if the asynchronous predicate is asserted. Conversely, if interval *i1* terminates before the asynchronous signal occurs, processing continues with the operations bound to interval *i2.* Many asynchronous predicates may be bound to the same interval, provided that they are mutually exclusive. Also, the interval that the asynchronous predicate is bound to may itself be hierarchically defined as a sequence of shorter intervals. The asynchronous predicate binding, and the associated branching behavior, applies to all constituent intervals.

For either asynchronous or asynchronous behavior, in the DDS the term *predicate* refers to a value or combination of values that will result in a particular branch

being taken. The predicate to be used is derived from the specification as follows. Some values are used directly as predicates. In particular, if all bits of a value are used to determine the branching behavior, that value may be used directly as a predicate. Examples include a one bit synchronous input whose value determines a two-way branch, or the *and* of two one-bit inputs that is used to determine a four-way branch. If not all combinations of the input bit values are significant, however, it may be more efficient not to use it directly as the predicate.[3.8] To illustrate, consider a processor that is looking for its address on a 32-bit bus. A two-way branch is made based on the success or failure of a comparison between the value on the bus and the address of the processor. Instead of using the 32-bit value as the predicate, the output of a compare operation in the data flow graph between the value on the bus and the address of the processor is used.

3.3.4.3 Loops

Loops, which may be nested, are represented in the CT model using two subscripted point types: a *begin-loop* and an *end-loop*, between which are located the events and intervals of the loop, all identified by the same subscript.[3.9] Loop exits are represented by the branching constructs described above. Figure 3.4 illustrates the DDS form of the SLIDE statement:

```
WHILE count DO
BEGIN
records = records + 1 NEXT
END
```

Count is a synchronous input. The glue arc, g, is required because the semantics of the begin loop point do not allow multiple outgoing arcs. The operation *inc*, of type *increment*, is represented in the DF Model, along with its input and output values. The index j defining the loop in the CT Model is also attached to the *inc* node, as well as to all its inputs and outputs. Incremented indices on the output values of data flow operations, such as $records_{j+1}$, indicate that these

[3.8] It is possible to define an *otherwise* branch from an *or* point, which means that not every combination of bit values must be individually specified.

[3.9] For simplicity, subscripts are sometimes omitted within figures.

values are used as inputs of the operation during the next loop iteration. (Other implementations of the DDS indicate indexed variables with the *j-1, j* pair of indices instead of *j, j+1.*) The condition *count* serves as the predicate on the or-fork point's two arcs. If the predicate is true, arc $i1_j$ is traversed. Because *inc* is bound to this arc, it is executed. The loop index *j* is then incremented and the loop is reentered. If the predicate is not true, arc $i2_j$ is traversed and the loop is exited. Note that because the operation is bound to an arc following the or-fork point, if the predicate is initially false, the operation is never performed. If instead *inc* is bound to an interval preceding the or-fork point, the effect would be that of a *Repeat "inc" until " count"* statement, in which the operation would be performed at least once.

3.3.4.4 Process Initiation, Priority, and Interrupts

Processes begin execution when the initiating conditions become true. These conditions may be either synchronous or asynchronous. To represent initiation for a set of processes, any number of which may be executing simultaneously, an *and* point is used, with one branch for each process.[3.10] An example for asynchronous initiating conditions is shown in Figure 3.6. Each branch of the *and* point begins with a *glue* arc followed by a *begin-loop* point. The arc following each of these points is associated with an asynchronous predicate related to the initiating condition for one of the processes. The binding for *Process1* to arc *i1* is shown in the figure. If and when the rising edge of predicate *init* is detected, a branch is made to the first point of the initiated process, *or_j*. This does not terminate execution of events on other arcs of the *and* point, however. The last point of *Process1* is the *end-loop* point *e1*, which corresponds to the *begin-loop* point *b1*. This defines a loop, the first part of which is a wait for the initiation conditions of *Process1*, and the last part of which is the process itself. This specifies a repeated execution of *Process1*, conditional on the assertion of init. If the initiating condition for any other process becomes valid simultaneous with or subsequent to initiation of *Process1*, the associated process is initiated at that time.

[3.10] The initiation of processes may also occur in parallel with other, unrelated events; these events would be bound to other arcs.

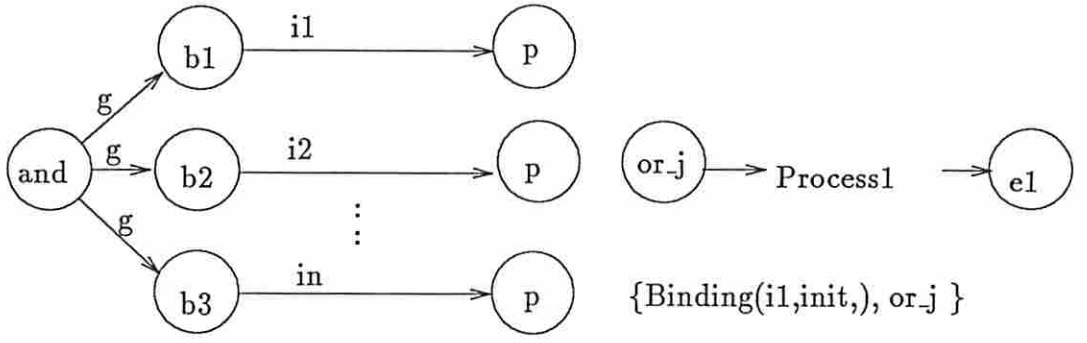


Figure 3.6: Parallel Process Initiation, Asynchronous Conditions

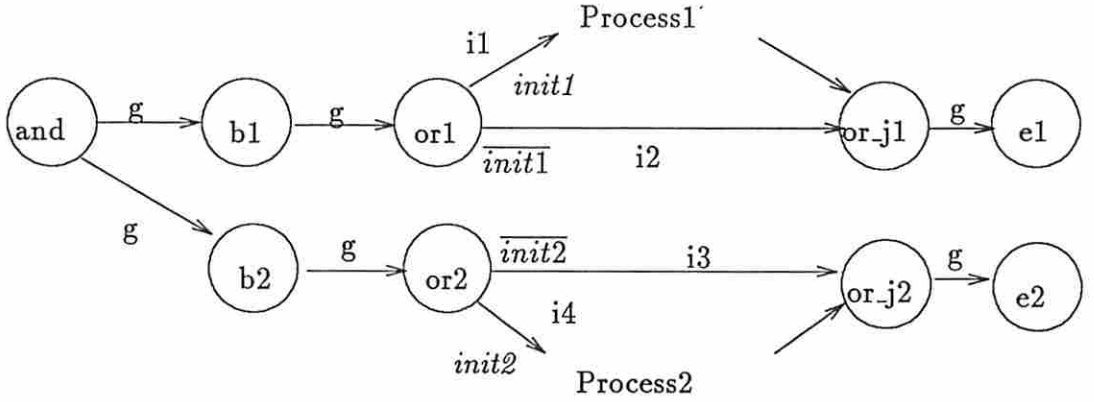


Figure 3.7: Parallel Process Initiation, Synchronous Conditions

If the initiating conditions are synchronous, then the *or* branch and associated synchronous predicate is used to represent the behavior in DDS. An example is shown in Figure 3.7. In this figure, the initiating conditions for two processes are checked. The *begin-loop* point of each branch of the *and* point is followed by an *or* point testing for the synchronous initiation conditions of one process. In the case of *Process1* for example, if *init1* is true when tested at point *or1*, then the branch *i1* is followed and the process begins execution. The last arc of *Process1* points to *or_j1*, which leads to the *end-loop* point *e1*, and another iteration of the loop starting at *b1*. If *init1* is not true when tested at point *or1*, then the branch *i2* is followed, which leads directly to *or_j1*, another loop iteration, and a check on the initiation condition for *Process1*. A similar description applies to the behavior associated with *Process2* on the second branch of the *and* point. This representation is used when any number of independent processes may be executing simultaneously.

In many cases, the number of processes executing simultaneously is limited by the specification. Several processes may be competing for a single resource such

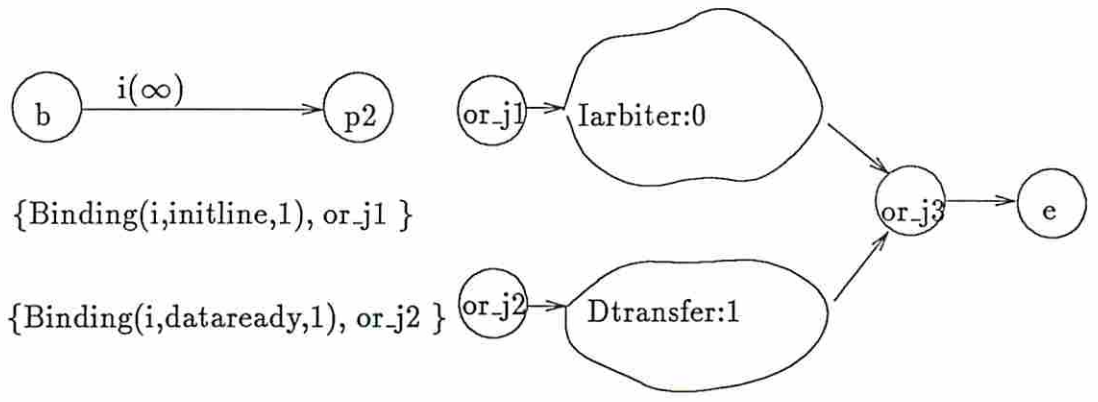


Figure 3.8: Priority Process Initiation, Asynchronous Conditions

as a bus, for example. In that case, an arbitration mechanism, or priority scheme, must be set up to determine which process may execute if the initiation conditions for several are valid simultaneously. As in the case for simultaneous execution, initiation conditions may be synchronous or asynchronous. An example for asynchronous initiating conditions is shown in Figure 3.8. In place of the *and* point and parallel branches, there is a single arc representing the time interval during which the initiation conditions of all processes are checked. This is represented by binding a set of asynchronous predicates to the same interval arc, *i*. The system waits during interval *i* for the assertion of one of the asynchronous predicates *initline* and *dataready*, which are bound to the interval. If *initline* takes on the value 1 processing branches to point *or_j1*, to which are bound the operations of the process Iarbiter. If instead, dataready is asserted, a branch is made to point *or_j2*, which defines the start of process Dtransfer:1. If more than one asynchronous predicate is bound to a single interval as in this example, they must be *mutually exclusive*. Therefore we must define a priority function on two non-mutually exclusive initiation conditions, so that if both are true simultaneously, only the predicate associated with the higher level process will be asserted. Priority is defined in the Data Flow Model using either gates or a specially defined priority encode operation as described in Section 3.3.2.

If the initiating conditions are synchronous, then a single *or* branch and associated synchronous predicate are used to represent the behavior. The synchronous

predicate may directly define the priority function. Take as an example three prioritized processes, with initiation conditions *init1, init2* and *init3*, where *init1* has the highest priority. In this case, the predicates bound to the three conditional branches would be: *init1*, $\overline{init1} \cdot init2$, and $\overline{init1} \cdot \overline{init2} \cdot init3$. If the initiating conditions are complex, a priority encode function is defined in the data flow graph. Only one output of the multi-output function is valid at a time, corresponding to the highest level initiation condition that is true. An example derived from the *priority-chain* of Section 3.2.3.3 is shown in Figure 3.9. Details of three of the inputs to *prior-encode*, which is a data flow *instance* of the operation type, *priority-encode*, are shown. By definition, only one of the values *c1* through *c5* is true at a time; therefore they can be used as predicates governing the conditional branch of the CT graph shown at the bottom of the figure.

Interrupts occur when higher priority processes or signals are allowed to preempt and terminate other processes. The interrupt may happen at any time during processing, unless it is *disabled.* It may be recalled from Section 3.3.4.2 that the interval to which an asynchronous predicate is bound may be hierarchically composed of *lower level* intervals and points. We can use this feature of the DDS to conveniently represent the branching action related to interrupts. An asynchronous predicate representing the interrupt condition is bound to an interval arc hierarchically composed of the lower priority process's specification. All of the lower level intervals *inherit* the asynchronous predicate and its branching effects. If the predicate is asserted during any of these intervals, a branch is made to the CT graph of the interrupting process or routine. Therefore the low priority process terminates. If the interrupt can be disabled, the disabling condition is included in the definition of the asynchronous predicate. Any number of interrupt levels can be conveniently represented this way. The introduction of new interrupts and priority definitions will require only minor modification to the graphs of processes.

Figure 3.10 demonstrates this representation. Dtransfer uses the bus, and is at a lower priority level than Iarbiter, which also requires use of the bus. If the initiating conditions of Iarbiter become true while Dtransfer is executing, execution of Dtransfer should be terminated and Iarbiter initiated. This is represented by representing Dtransfer at the highest level by a single interval, *i1.* Bound to this

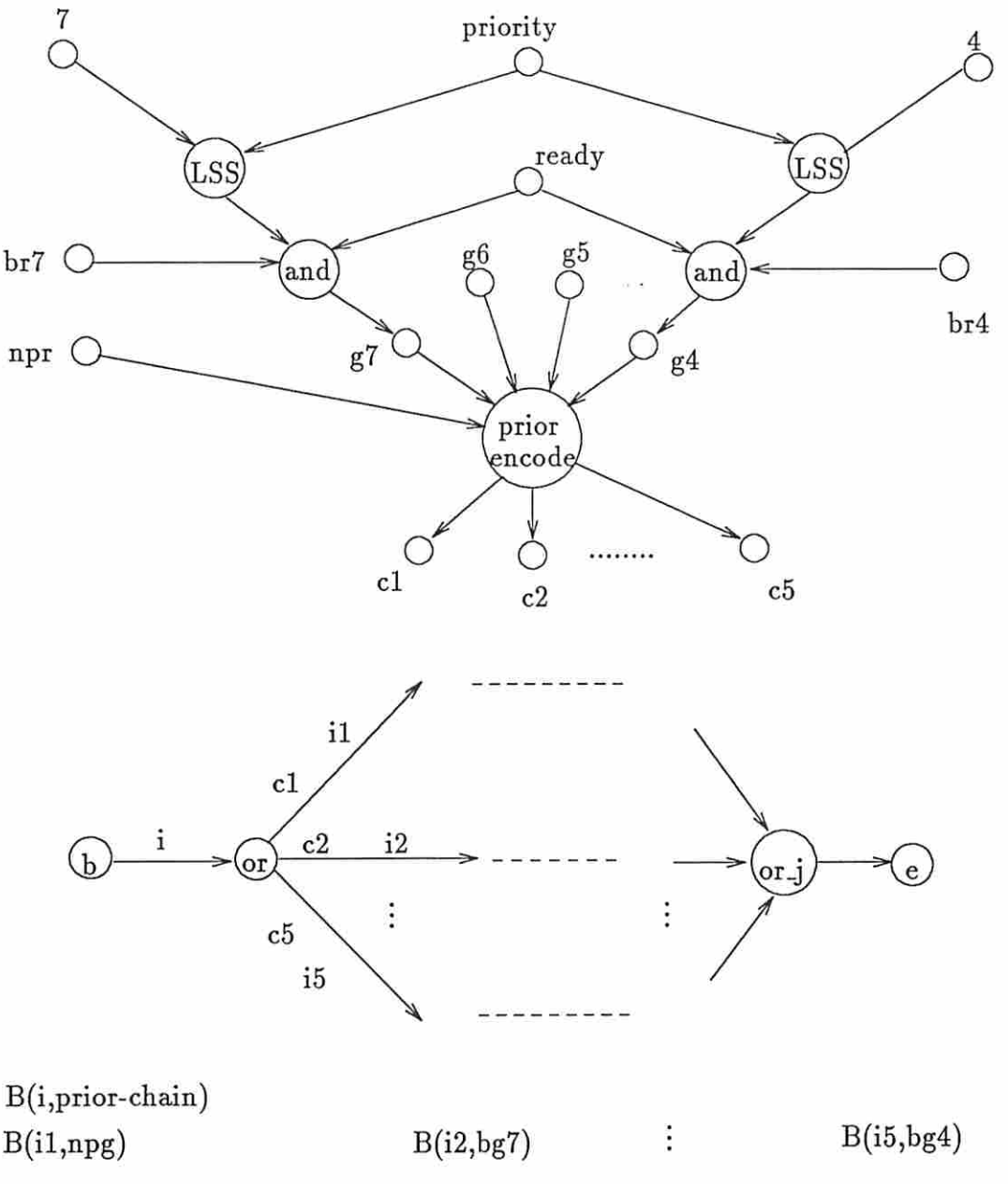


Figure 3.9: Priority Process Initiation, Synchronous Conditions

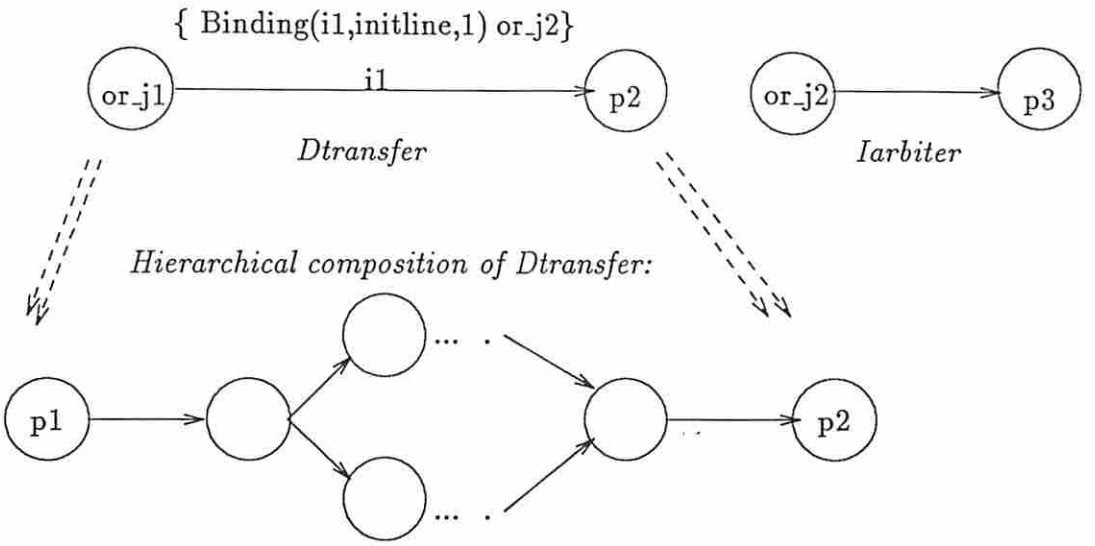


Figure 3.10: Process Priority

interval is an asynchronous predicate representing the initiation conditions for Iarbiter. Interval *i1* is hierarchically composed of the behavior of Dtransfer, shown at the bottom of the figure. All lower level arcs of Dtransfer inherit the asynchronous predicate binding of *initline*. If Dtransfer is initiated before Iarbiter, processing begins at *p1*. If initline is asserted before Dtransfer terminates normally, however, a branch occurs to the first node of the Iarbiter process.[3.11] This represents the abnormal terminal, or interruption, of Dtransfer.

3.3.5 Timing Behavior

3.3.5.1 Timing Constraints

To represent general time constraints between events, such as *Operation X must occur within z seconds after operation Y,* a *time constraint* arc is used in the CT graph. Time constraint arcs may begin and end at any type of point, but are not used as connectives between otherwise unconnected graph segments. The arc specifies only timing information; therefore operations may not be bound to it. The information may take the form of an equality or an inequality specifying a minimum and/or maximum time. A time constraint imposed between the *begin*

[3.11] If some interrupt sequence is to be executed before the actual branch occurs, intermediate nodes and arcs will define the necessary processing.

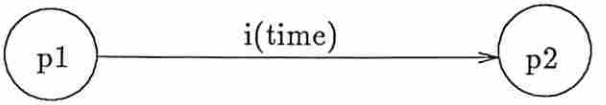


Figure 3.11: Simple Time Interval

and *end* points of a loop refer to the time of a single execution of that loop. Time constraints that enclose the entire loop construct include all iterations of the loop. Constraints cannot be placed across conditional or parallel branches.

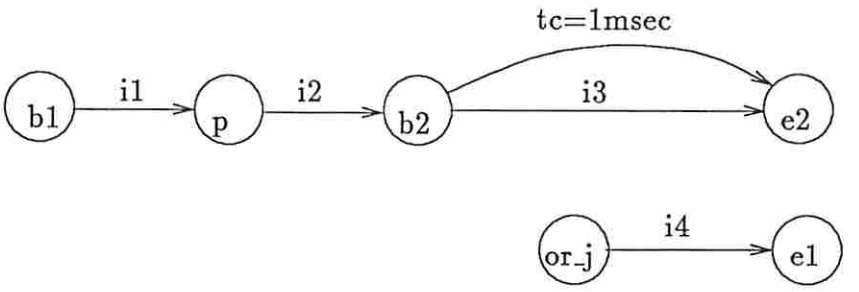
3.3.5.2 Time Intervals

Time intervals of specified values are represented in the CT Model by assigning a length to an interval arc, as shown by *time* in Figure 3.11. If any outputs are to be asserted during this interval, the appropriate output bindings are made to it. This corresponds to the SLIDE statement: *Delay "time"*, where the length of the interval arc is *time*.

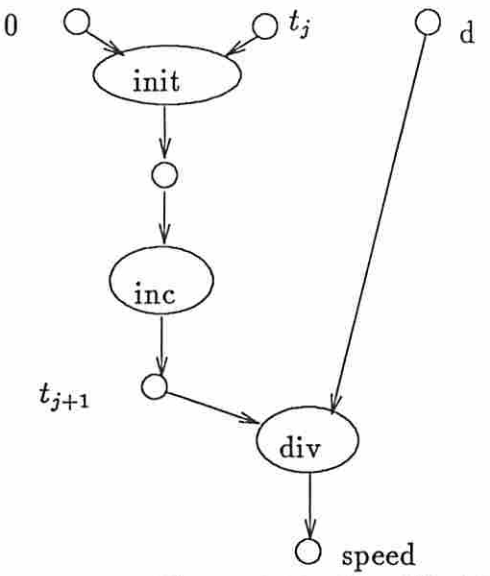
To represent the measurement of a time interval bounded by two events, a variable representing the time is defined in the DF Model. An operation incrementing this variable by one is contained in a loop that is initiated by the first event and terminated by the second event. In addition, a time constraint equal to a single time unit is imposed on the loop. When the loop terminates the variable will contain the number of time units that elapsed between the occurrence of the first and second events. An example is shown in Figure 3.12. This is a simplified specification for a drag strip controller [Com84]. During the first interval, *i1*, the red light is asserted, followed by assertion of the yellow light in *i2*. [3.12] The deassertion of the yellow light following *i2* is a synchronous event that signals that the time count should begin. The awaited event to terminate this count is the assertion of the asynchronous input *left.fin* that signals that the left car in the race has crossed the finish line. Interval *i3* is enclosed in a loop. During each iteration, the green light remains asserted and the data flow operation *incr* of type *increment* is executed. This operation is constrained to a duration of 1 msec. by the time constraint arc *tc*. When *left.fin* is asserted a branch is made to point *or_j*, as specified by the

[3.12]The *init* operation bound to *i2* is a specially defined DDS primitive to initialize *t* to *0* preceding the count.

CONTROL and TIMING MODEL



DATA FLOW MODEL



BINDINGS:

i1: red light

i2: yellow light, init

i3: green light, inc

left.fin= 1 ⇒or_j

i4: divide

Figure 3.12: Simplified Drag Strip Controller

asynchronous predicate binding to i3. During interval *i4*, the speed of the left car is calculated by dividing the race distance d by the time t, whose index reflects the loop iteration number. Because the operation incrementing t is constrained to last 1 msec., t will give the time in msec. This DDS graph corresponds to the SLIDE specification:

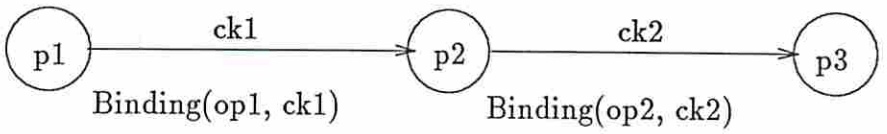


Figure 3.13: Synchronous Behavior with Clocking

```
t = 0
green.light← 1;
WHILE NOT left.fin DO t = t + 1 (L: label1) NEXT
speed=d÷t
TIME label1:  1msec
```

3.3.5.3 Synchronization

Interface behavior often involves *periodic signals* and the relationship of signal changes to clock levels is often a crucial part of that behavior. In a DDS representation of a specification containing synchronous signals, clock periods are considered to be a special form of time interval. They will be designated by the symbol *ck* and associated with a *length* specifying a time period, if known. The CT Model of a synchronous design would therefore include a sequence of clock period intervals. The synchronization of operations and signal events to the clock period is represented by bindings to the intervals (Figure 3.13). For those applications in which clock levels are related to system events, we introduce a distinction between clock phases by defining the interval subtypes *low* and *high*. For detailed behavior related to edges of the clock signal, such as inputs that change on a rising edge and conditional outputs asserted on the falling edge, special data flow values *rise* and *fall* can be defined. This corresponds to the SLIDE key words `SYNCF` and `SYNCR` and the SLIDE symbols \ and /.

The test for a sequence of values on a synchronous input line is represented in the CT Model by synchronous conditional branches. The specifications for a sliding versus nonsliding window vary in the nature of the branching. The example [Com84] given in Section 3.2.4.3 is repeated here to demonstrate the DDS representation for a nonsliding window bit-pattern search.

> If the sequence 10 occurs on the serial data line X, output Y should be asserted for 1 clock period before the next check is started. If 01

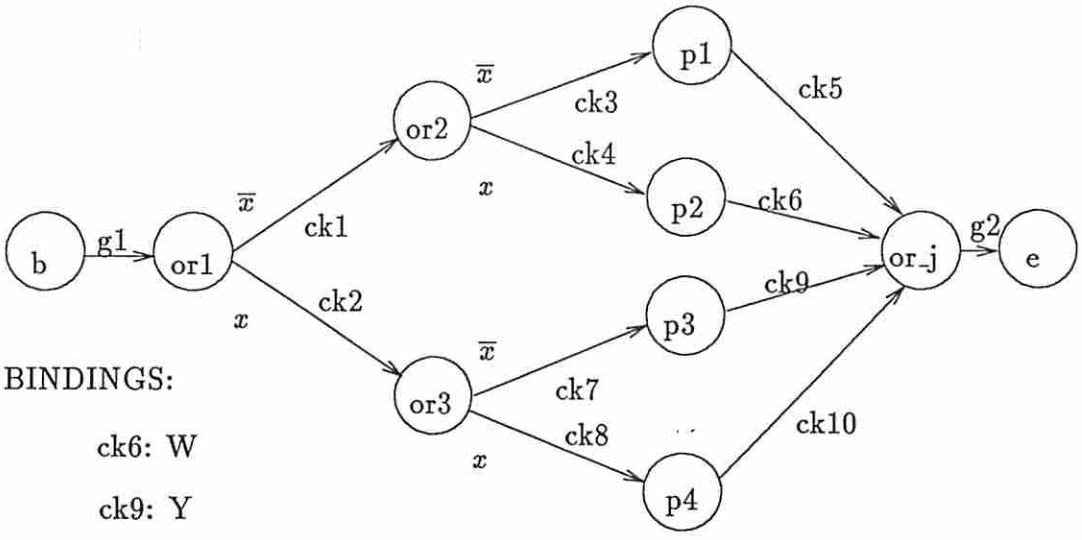


Figure 3.14: Bit-pattern search, nonsliding window

> occurs, output W should be asserted for one clock period before the next check is started. If 00 or 11 appear as the two sequential bits on X, no output should be generated and the next check should begin after a delay of one state time.

Figure 3.14 shows the DDS representation of this behavior. The *b* and *e* points define an outer loop that repeats the behavior indefinitely. All *ck* intervals represent clock periods. The numbers attached serve to uniquely identify the individual arcs and do not indicate different clocks. At each *or* point, the value of the synchronous 1-bit input X is checked, and a branch made accordingly. One path through the graph, $g1 \rightarrow ck2 \rightarrow ck7$, corresponds to the input sequence 1 | 0. The following interval, *ck9*, has an output binding defined for Y, representing the output called for in the specification. A similar path represents the input sequence 0 | 1 and the associated output, W. All paths through the graph are of the same length. If the sequence 11 is detected, for example, the loop iteration is not terminated until one more clock period has elapsed. This behavior corresponds to the SLIDE *seq* statement describing the same specification in Section 3.2.4.3.

The design of a DDS graph to recognize a sliding window bit pattern is basically identical to the design of an automaton to perform general pattern matching. The Knut-Morris-Pratt pattern matching procedure constructs a deterministic finite-state automaton that enters an accepting state if a subsequence of characters of

an input string matches the pattern. If a mismatch occurs, the automaton takes a "failure transition" back to the "best possible" state and continues pattern matching. The best possible state to recover from failure is found by determining the longest head of the pattern that matches a tail of the input string. The automaton is constructed using only the information in the pattern in time proportional to the number of characters in the pattern.

The first step in constructing the automaton is establishment of a sequence of m state transitions with $m + 1$ states, where m is the number of characters in the pattern. State j is reached when the first j characters of the pattern match the last j characters of the input string scanned. The failure transitions are constructed through use of a table of "failure links." The complete algorithm is not described here, but can be found in [Sta80].

This algorithm can be modified to automatically produce a DDS specification of the behavior given the bit pattern to search for. Since the translation from SLIDE to the DDS is done manually, however, this step has not been automated. To illustrate the process the second example given in Section 3.2.4.3 is repeated below and the DDS graph created to represent the behavior is shown in Figure 3.15.

> When the sequence *1,0,1,1* is detected on line X, the record count is to be incremented and the check for the next sequence begun immediately.

This behavior corresponds to the SLIDE *delay* statement describing the same behavior in Section 3.2.4.3. The path $ck2 \rightarrow ck4 \rightarrow ck5 \rightarrow ck7$ corresponds to a match of the incoming synchronous values on line X with the pattern. Bound to interval *ck7*, therefore, is the data flow operation *increment*, which increments the value *record-count* defined in the Data Flow Model. The path through ck1 is only one clock period long, however, and is followed when the first input bit does not match the first pattern bit. A new loop iteration will begin at b1 at the next clock period. If the first input bit is a 1 the following bit must be a 0 or the pattern does not match. If however, the next bit is another 1 we loop back to b2 until an incoming bit has the value 0. This way we can detect the pattern 1011 within the sequence 11011, for example. Similarly, if multiple *1-0* pairs occur in the input stream, a loop is executed at b3 until either a match occurs with a 11 sequence, or a 0 is read in as the next bit, in which case the pattern match begins anew at b1.

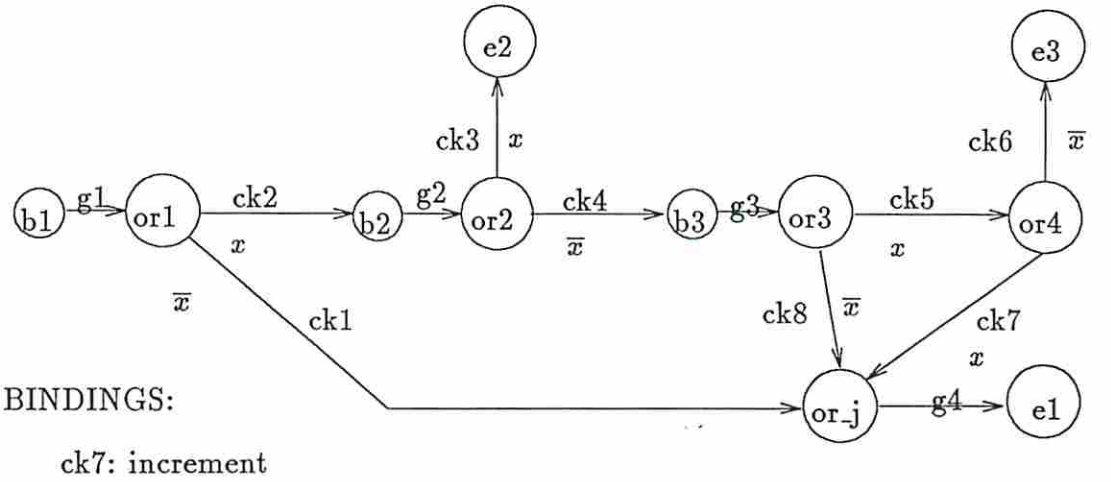


Figure 3.15: Bit-pattern search, sliding window

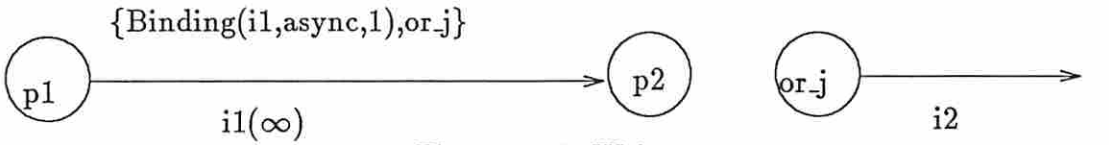


Figure 3.16: Waits

The synchronization of system behavior to signals other than the clock can be represented in the DDS using the asynchronous predicate binding to specify system behavior upon receipt of the synchronizing signal. In the previous section it was explained how branching behavior governed by asynchronous signals is represented through use of an *asynchronous predicate binding.* If the asynchronous signal is asserted anytime during the interval, a branch is made. To represent an indefinite wait for assertion of the asynchronous signal, the interval to which the predicate is bound is assigned an infinite time limit, and there is no succeeding arc. An example demonstrating the DDS representation of synchronization achieved by an indefinite wait (corresponding to the SLIDE statement *delay until async eql 1*), is shown in Figure 3.16. When the value *1* is detected on the line *async* during *i1*, a branch is made to point *or_j* and the events bound to *i2* are executed. If there are any outputs bound to *i1*, they are asserted as long as *async* is not asserted.

If synchronization is not achieved within a predefined period, the wait on the synchronizing signal may be terminated (timed out). This is represented by assigning a length to the interval to which the predicate is bound equal to the waiting time. Any outputs that are to be asserted during the wait are bound to the same

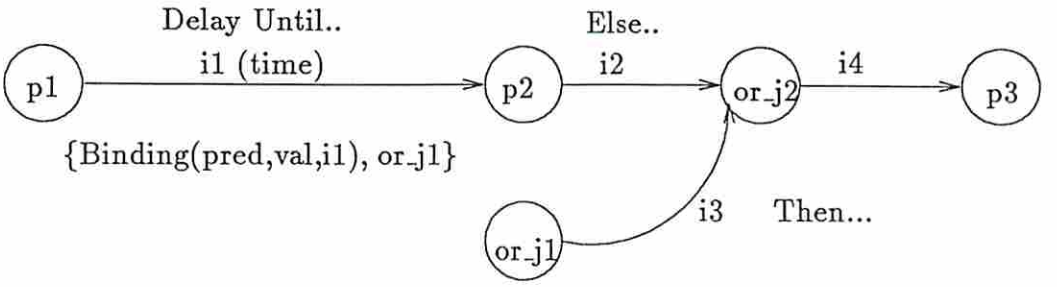


Figure 3.17: Time-outs

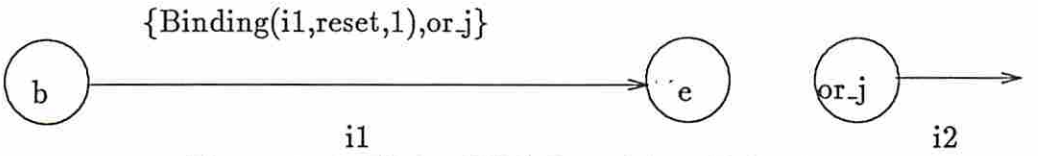


Figure 3.18: Waits, With Repetitious Behavior

interval as the asynchronous predicate. An example demonstrating the DDS representation for time outs, which corresponds to the SLIDE statement

```
DELAY time UNTIL async EQL 1 THEN statement1 ELSE statement2
```

is shown in Figure 3.17. The first interval *i1* has been assigned a time length. The asynchronous predicate is bound to this limited interval. If the time elapses without assertion of the proper signal a time-out occurs. If there are any operations conditional on a time-out, such as those in *statement2* of the *else* clause above, they are bound to *i2*. If no else clause is present, *p2* and *i2* are omitted. If the asynchronous predicate *is* asserted some time during *i1* then there is no time-out, and processing branches to *or_j1*. Any operations conditional on assertion of the asynchronous signal, such as those in *statement1* of the *then* clause above, are bound to *i3*. If no *then* clause is present, *i3* and *or_j1* are omitted and the system branches to *or_j2* when the asynchronous predicate is asserted. The last interval in this figure, *i4*, is bound to whatever operation follows the delay statement.

The specification may define some repetitious behavior to be executed during a wait on an asynchronous signal. To represent this in the DDS, the behavior to be repeated and the asynchronous predicate will be bound to the same interval, enclosed in a loop. The loop is exited when the asynchronous predicate is asserted. This is illustrated in Figure 3.18, with an example corresponding to the SLIDE statement: *WHILE reset EQL 0 DO statements. i1* and

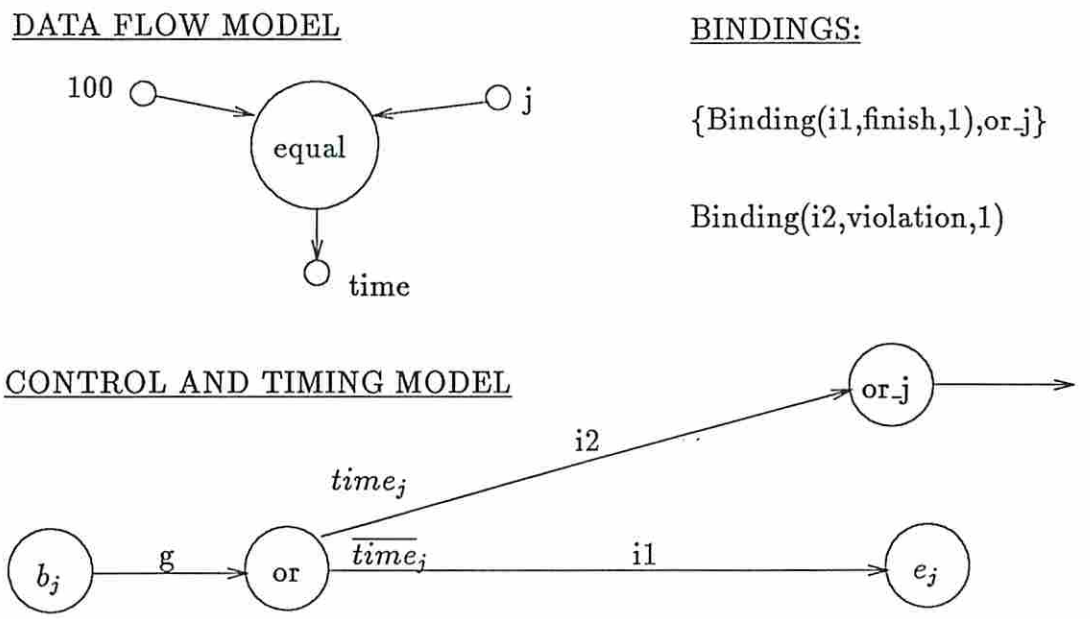


Figure 3.19: Waits, With Repetitious Behavior and Time Outs

The asynchronous predicate is the opposite of the *while* condition, since it defines the exit condition. Points b and e define the begin and end of a loop that stipulates that the *statements* are to be repeated. The loop has no exit other than that related to the assertion of *reset*, at which time a branch is made to *or_j*. The *statements*, whether outputs or operations, are bound to interval *i1* and are repeated once each iteration of the loop, until *reset* is asserted.

To introduce a time-out capacity for this representation, the loop index can be used as an exit condition. The time must be translated into the number of loop cycles, and before each loop is executed, the loop index is compared to this value. This is illustrated in Figure 3.19, with an example corresponding to the SLIDE statement:

```
WHILE (100) (finish EQL 0) DO (count = count +1)
ELSE (violation_1)
```

Here, *100* is the time-out clause, expressed in clock periods. In the figure, *time* is a data flow value, the output of an operation comparing the loop index, *j*, with the constant *100*. If the time limit has not been reached, the top branch is followed, and the operation bound to interval *i1* is executed, provided that the asynchronous predicate *finish*, which is also bound to *i1* is not asserted. If *finish* is asserted, a branch is made to *or_j*. If the time limit has been reached before *finish*

is asserted, the bottom branch is followed, and *violation* is asserted, as stipulated by the binding to interval *i2*.

3.4 Examples

3.4.1 Frame Counter Example

To further illustrate the representation of synchronous behavior, a small interface specification will be given. The description is first given in English, and then in SLIDE, followed by the corresponding DDS representation.

A data acquisition system sends serial data at a 1 kHz bit rate to a recording system. The data is in the form of 4-bit words. Several records, each consisting of a variable number of words, will be transmitted each time the data acquisition unit becomes active. Each record is separated from the following record by the frame separator word 1101 that will never be contained in a data record. A count line from the acquisition unit to the recording system will be asserted to signal that counting of the records should begin and will be deasserted to signal the end of record transmission. A clock signal from the data acquisition unit is available, and transmitted data changes on the negative transition of this clock. The count signal will always be asserted as the first bit of the first record occurs. The maximum number of records to be counted is 200.[Com84]

```
MAIN PROCESS FRAME-COUNTER
SYNCF D(), C()
CLOCK 1 kHz
REGISTER COUNT(7:0)
INIT COUNT-WORDS:1 WHEN C
PROCESS COUNT-WORDS
BEGIN
WHILE C DO
        BEGIN
        IF D SEQ 1:1:0:1 THEN COUNT = COUNT + 1 END NEXT
        END
END
```

Figure 3.20 shows the Control and Timing Model and Data Flow Model with associated bindings for this specification. To simplify the example, an outer loop to reinitialize *count* has been eliminated and also, loop indices have been omitted. C is the count line and D is the data line. Arcs labelled with the symbols *hi* and *lo*, symbolizing high and low clock phases, are used to indicate that values of C and D are not asserted until the falling edge of the clock. At the first *or* point C is tested: if it is asserted the body of the loop is entered. Otherwise, the body is skipped and the loop is restarted at the next clock period to again check the value of C. At each *or* point in the graph a check is made on the value present on line D during the low phase of the clock. Whenever a particular value of D indicates that the current word is not the frame separator, the cycle of four clock periods is nevertheless completed before the loop is restarted. The branch followed in that case employs *ck* arcs, which indicate full clock periods and make no distinction between the low and high clock phases, since no tests of input values need to be made.

When the frame separator word is detected, an *and* point is used to indicate that two parallel actions are to take place. The arc *lo7* completes the current cycle and is directed to the last point of the loop, indicating that the next cycle is to begin. The second arc *i* is bound to the increment operation in the Data Flow Model that updates *count*. Because *INC* must terminate before the next record is detected, a time constraint is attached to its interval. On the assumption that each record consists of a least one word in addition to the frame separator, the constraint is set at seven clock cycles. Because the clock rate is specified, each type of clock arc will have a length associated with it. Also, since the maximum size of the value *count* is given, it can be bound to a 16-bit register in the structural model.

3.4.2 Arbiter Process

An example that illustrates several of the preceding concepts is taken from the Arbiter Process description of the Unibus. Its SLIDE description is as follows:

```
IF priority LSS 7 THEN
BEGIN
```

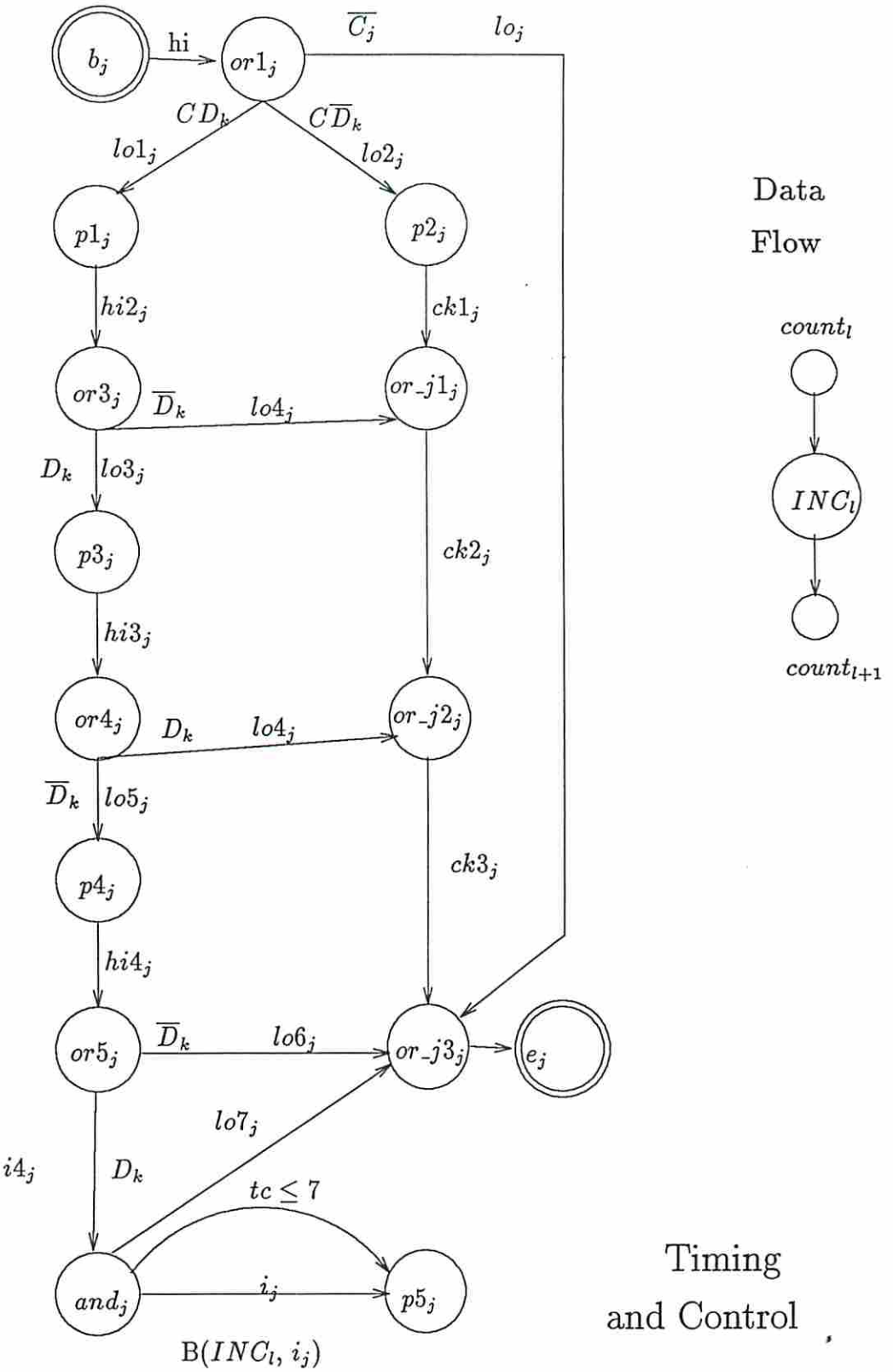


Figure 3.20: Frame Counter Example

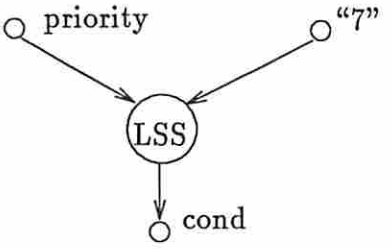
```
bg7_1;
DELAY 5000 UNTIL sack EQL 1 NEXT
bg7_0;
END
```

Figure 3.21 gives the DDS representation of this specification. Three models are necessary to capture the description. Following the timing graph we see that the boolean operation *LSS* is bound to the first time range, *i1*. This operation computes the value *cond* that functions as a synchronous predicate in the conditional transition that occurs at the *or* node of the Control and Timing Model. Interval *i2* is followed when *cond* is true, otherwise interval *i4* will be traversed. If *cond* is true, this indicates that *priority* is less than or the same as 7, which means that bus grant 7 (*bg7*) should be asserted. The bus grant line should be asserted for a maximum time of 5000 ns, or until the selection acknowledge (*sack*) is received. This is indicated by two bindings defined on *i2*. The first, B(i2,bg7,1), indicates that a value of 1 should be placed on bg7 during this time interval. The second, {B(i2,sack,1),or_j}, defines an asynchronous predicate, *sack*, and indicates that if and when sack is asserted, processing should immediately branch to point *or_j*. The time length of 5000 assigned to interval *i2* indicates that time range *i2* should not exceed 5000 ns, and processing should proceed from point *or_j* after that interval has passed. The binding attached to interval *i3* indicates that *bg7* should be deasserted.

3.5 Summary

This chapter has described a formalism for the representation of interface behavior that can be used for synthesis by a design automation system. The behavior of interfacing processors was studied and analyzed, and a classification was established. This classification benefits the synthesis work by characterizing the problem, thereby establishing the types of behavior that must be represented in the specification and identified by the synthesis system. The hardware descriptive language SLIDE was chosen as the specification language and some extensions made based on the analysis of the problem domain characteristics.

Data Flow Subspace:



Structural Subspace:

carrier: bg7, sack

Timing and Sequencing Subspace:

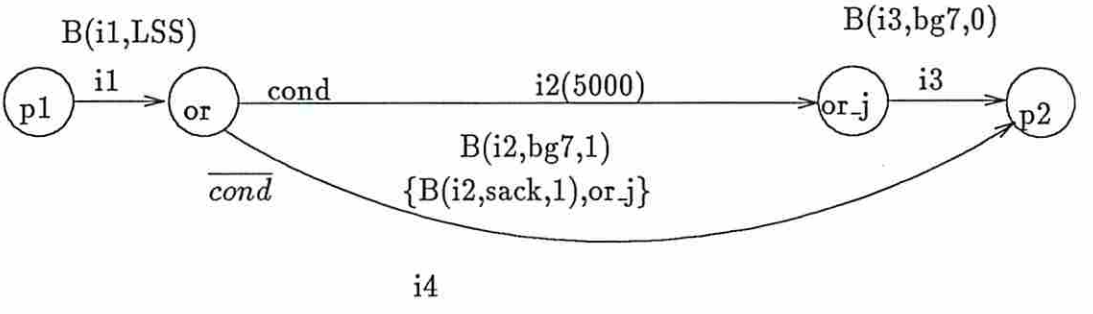


Figure 3.21: Partial Description of UNIBUS Arbiter in DDS

The graphical representation called the Design Data Structure was chosen for the internal representation of behavior. The Design Data Structure can represent the many facets of interface behavior in a unified way, including timing constraints, synchronous and asynchronous signals, control flow, and data manipulation. Its descriptive power is more complete than some other formalisms in use, including event and annotated data flow graphs. The same representations described here can be used by data path synthesizers to capture more complex timing information than is typically handled and to separate control from data manipulation information to produce cleaner data flow graphs. A determination was made of how to represent the different classes of interfacing processor behavior, based on the classification in Section 3.1. Some extensions and modifications to the DDS were made to improve this representation.

This chapter has established the representation methods used to capture the desired behavior in a specification. The specification is used as the input of a synthesis system that derives from it an implementation of the behavior. The next chapter describes the synthesis methods developed to create a state table specification of the control for interfacing processors, and the implementation called CONSPEC.

Chapter 4

Goal-Based Design Management

In this chapter we describe our design management strategy that uses explicit goal-based reasoning. In the first section we define some concepts and give motivation for the use of explicit goals in design. This is followed by sections on CONSPEC's system architecture, goal detection, plan selection, the handling of interactions between design tasks, and plan execution.

4.1 Explicit Goal-Based Reasoning

Design objectives are defined explicitly by establishing goals to represent each objective. Explicit representation permits the use of a planning strategy for the solution of multiple design objectives. During planning, solution techniques are chosen for each goal and a sequence of steps is mapped out to achieve system goals before any steps are actually executed. Goal-based design management and planning are particularly advantageous when interactions of various kinds exist between the goals. In this way difficulties can be anticipated and wasteful actions avoided.

During control synthesis interactions exist between many design decisions. Figure 4.1 uses a partial DDS Control and Timing graph specification to illustrate several interactions. The and and and-join (and-j) points signify parallel forks; a loop is represented by a pair of begin (b) and end (e) loop points; time constraints are shown by solid, curved lines annotated with constraints on the execution time (t). Intermediate points and intervals are omitted, as represented by the dashed lines.

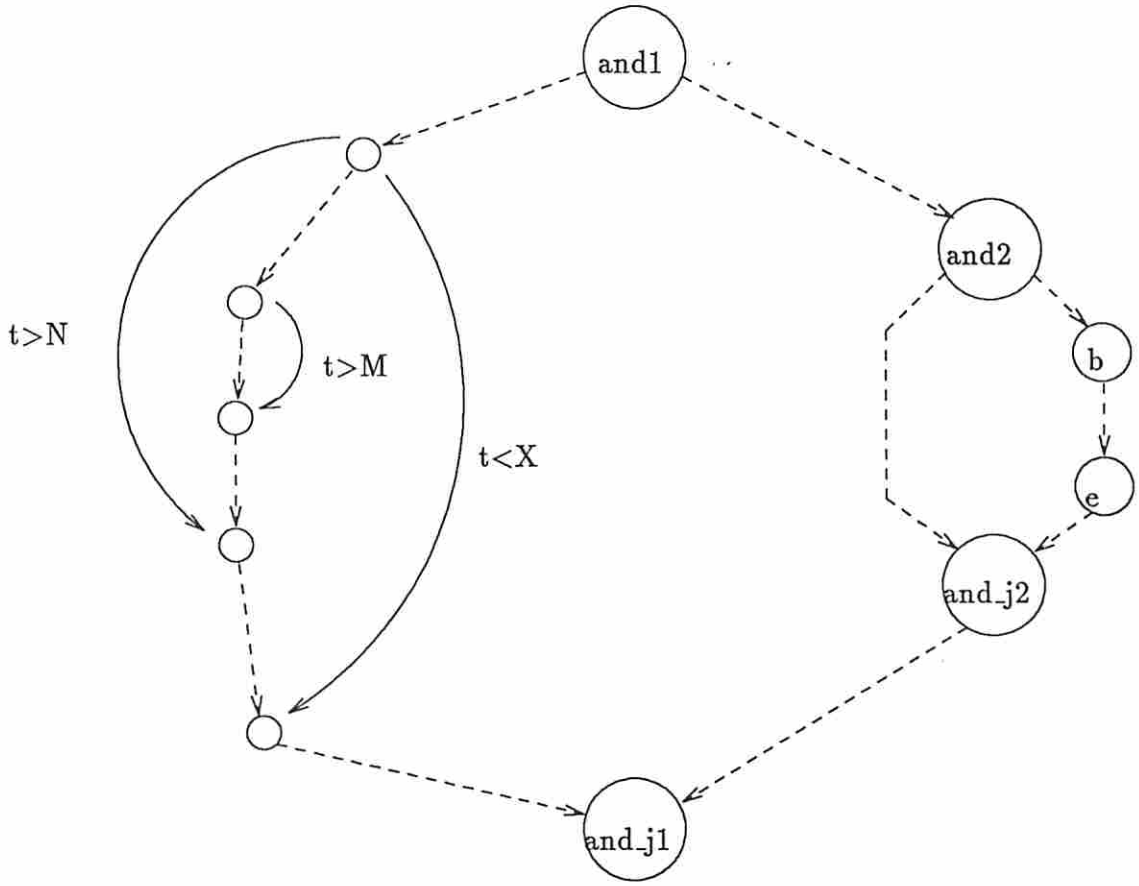


Figure 4.1: Interactions Between Design Tasks

The first interaction we consider is between the two minimum time constraints, one of which, $t > M$, is nested inside the other, $t > N$. CONSPEC satisfies minimum time constraints by the addition of delays if necessary. If we schedule satisfaction of the nested constraint before the outer constraint, any delay added to satisfy the inner constraint will also contribute to the satisfaction of the outer constraint. This is a case of positive interaction between design tasks. Conversely, if the outer constraint is satisfied first any delay added does not contribute to satisfaction of the nested constraint unless it is added to a region of the graph where the constraints overlap. We can easily take advantage of the positive interaction and reduce the total amount of delay added by scheduling satisfaction of the nested constraint first.

For this specification it is particularly important to add the least amount of delay to satisfy the minimum constraints because of a second interaction between them and the maximum constraint, $t < X$. Because both minimum constraints are enclosed within the maximum constraint, we should add as little delay as possible to satisfy the minimum constraints. Adding more than absolutely necessary may preclude satisfaction of the maximum constraint. In addition, the satisfaction of the maximum constraint by reduction of the execution time may also result in violation of the nested minimum constraints. Overlapping minimum and maximum time constraints are an example of negative interaction between design tasks. In general, goal solution should consider negative interactions with other goals, since the solution of one may unwittingly violate another.

A third interaction arises between implementation of the loop and the parallel fork at *and2*. CONSPEC uses two implementation techniques for parallel forks. First, they can be implemented using a separate state machine for each branch. The branch machines are initiated by a signal from the principal state machine when the parallel fork is reached. The principal machine loops until termination signals are received from all branch machines and then continues with processing following the parallel fork. For our example in Figure 4.1, the parallel forks at *and2* would be implemented by two finite state machines, one for each branch. The second implementation technique for parallel forks is to merge all branches state by state and implement the combined behavior using the principal machine. Two requirements for this are (1) the absence of internal loops and (2) simplicity of

conditional state transitions on the branches. If these requirements are not met, the number of states and next state transitions of the single machine implementation will explode. In that case the multiple machine implementation is preferable since fewer states are required.

The internal loop on the right branch of the nested parallel fork at *and2* mandates the use of two branch machines. Because of the state loop needed to implement the handshaking protocol between the main and branch machines for this fork, we will also be unable to merge the branch machines of the outer parallel fork at *and1*. If the loop on the right branch of *and2* has a constant number of iterations and can be unwound first, however, it may be feasible, depending upon transition conditions, to implement the entire specification using a single machine instead of four separate ones. This is a case of one simplifying modification (loop unwinding) creating the conditions for or eliminating barriers to another simplification (single machine implementation).[4.1]

A fourth interaction in this example exists between the satisfaction of time constraints and the merging of parallel branches. If a time constraint applies to only one branch of a parallel fork, as in our case, it is preferable to satisfy the constraint before attempting a merge of the branches. This simplifies maximum time constraint satisfaction, since only the events and operations on a single branch are involved. It also reduces the performance impact of minimum time constraint satisfaction, since any added delays affect only the operations on a single branch of the fork.

Certainly not all design specifications will have problems arising from such interactions, and not all potential interactions have actual negative or positive effects (not all parallel branches can be implemented by a single state machine, for example). Nevertheless, in the general case interactions of this sort can and will arise; a design system with knowledge about interactions can create better and more intelligent designs in such instances than a system that assumes all its actions are independent.

One advantage of a goal-based planning strategy is that interactions are represented in formal ways, traditionally through the use of pre and post-conditions.

[4.1] Loop unwinding has not been implemented in CONSPEC, but is used to illustrate the impact of interactions between design decisions.

This results in greater modularity in the knowledge representation than would the "hard-coding" of potential interactions, leading to greater ease in system extension and modification. The explicit representation of goals also makes it easier to reason about goal failure and to report failure to the user. CONSPEC takes advantage of this to perform a limited amount of failure analysis, as discussed in Section 4.6.

In Chapter 3 we described some planners that deal with particular kinds of interactions. Of particular importance for this research, however, is the Theory of Planning developed by Wilensky [Wil83] for natural language understanding and common-sense reasoning.

Wilensky's planner detects its own goals from the input specification. There may be more than one solution, called a plan, possible for each detected goal. One plan for each goal is chosen, followed by meta planning to detect and account for positive and negative interactions between the chosen plans. Detection of an interaction leads to the establishment of a resolve-interaction goal, for which a plan is also chosen. Resolve-interaction goals and their associated plans embody the system knowledge of what conflicts potentially exist and how to attempt a resolution or tradeoff.

Wilensky's work is relevant to design for several reasons. First, several solution strategies or plans may exist for a single goal; a primary task of the planner is plan choice for each goal. This is unlike many planners such as Noah, which assume a single operator choice for each primitive goal. If multiple choices exist, Noah tries all possible combinations. Most design problems are characterized by multiple implementation possibilities that would lead to combinatoric problems using Noah's method.

Second, interactions between goals are handled using the general power of the planning agent, so the actions (plans) taken to resolve interactions can be arbitrarily complex. Domain-specific goal interaction types and solution techniques can be defined. For example, in dealing with the interaction between minimum and maximum time constraints, the system can take advantage of the fact that it is far easier to increase execution time than it is to reduce it. This is significant because domain-independent planners are weak with regard to interaction handling [Cha87]. A third important feature of Wilensky's planning theory is that individual plan choice can be made based on more than one goal at a time, that

is, interactions between goals can influence plan choice or suggest new plans for the solution of elementary goals. These three strategies have been adapted and incorporated into CONSPEC, as can be seen in the following sections.

4.2 System Architecture

The design is executed in two phases as shown in Figure 4.2. During the first phase a design manager calls a procedure to derive an initial state table. During the second phase the design manager invokes four design agents that may make modifications to the state table. The agents include a goal detector that examines the state table for optimization and constraint goals, a plan proposer that chooses a plan for each detected goal, an interaction handler that identifies interactions between plans and takes corresponding actions, and an executor that executes the final plans.

The design manager cycles through the design agents in the order goal detector, plan proposer, interaction handler, and executor. Interactions are dealt with after plans have been proposed for all goals because different plans for the same goal type may have different impacts on the design. Interactions are detected and treated before plans are executed, however, allowing us to choose different plans, arrange the order of plan execution, or eliminate individual plans to deal with potential interactions before they can affect the design. In this way we may succeed in avoiding negative interactions and the resulting failure of individual goals. Similarly, we may be able to take advantage of positive interactions between the plans for individual goals.

The CONSPEC organization has a blackboard-like architecture [Nii86]. As shown in Figure 4.3 the design agents access design knowledge in the form of declarations and procedures contained in a knowledge base, and read from and write to a global data base, or blackboard. In this section we describe the design phases, knowledge base structure and contents, and blackboard characteristics.

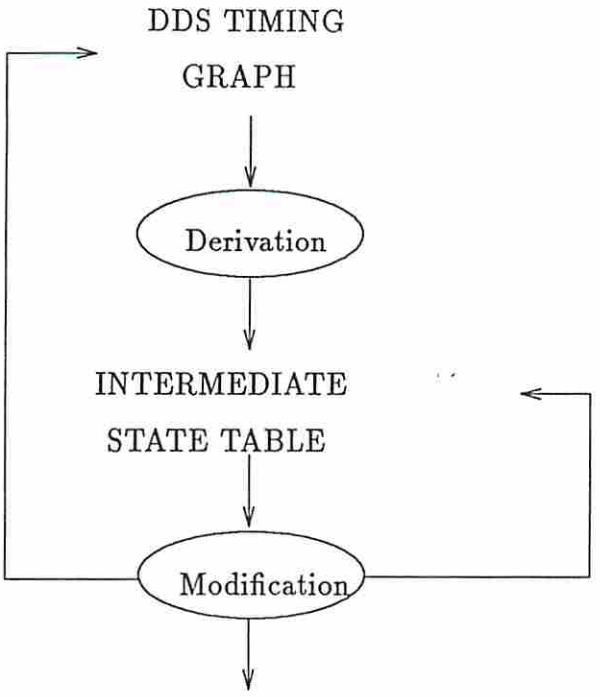


Figure 4.2: Design Phases

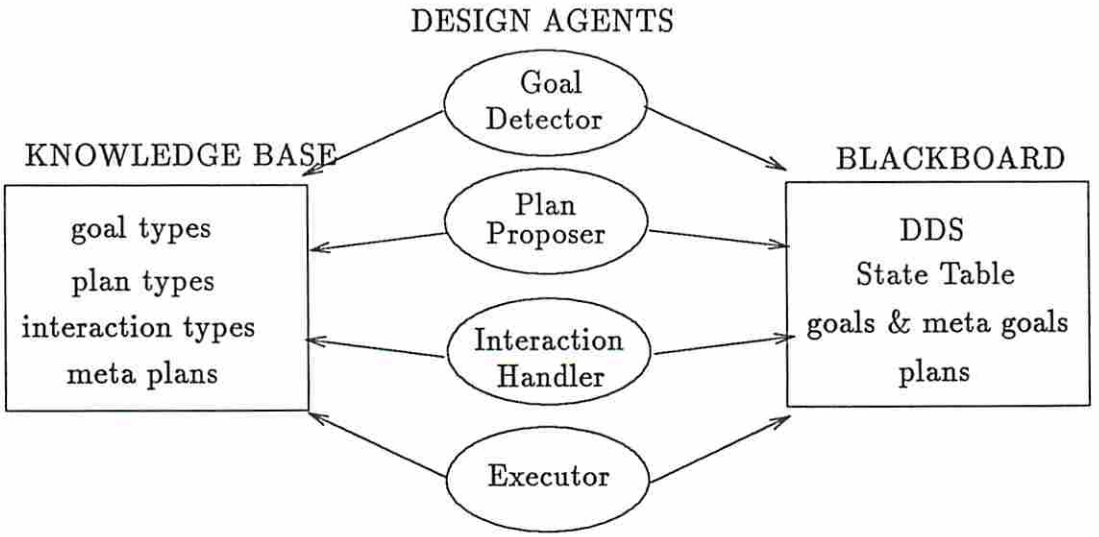


Figure 4.3: CONSPEC Architecture

4.2.1 Design Phases

Figure 4.2 shows two phases, derivation and modification, through which the design passes. During the derivation phase CONSPEC creates a state table implementation from a DDS Control and Timing Model graph specification, using algorithmic techniques. A procedure performs the derivation state by state during a depth-first traversal of the DDS timing graph. During the derivation, described in detail in Section 5.2, three general tasks are involved. First, CONSPEC determines which behaviors described in the graph should be included in a single state and the number of states required to implement the behavior. Second, it decides which outputs should be produced during particular states. Third, it determines which, if any, input values affect the conditions under which outputs are produced and next state transitions occur for each state. Parallel forks are handled by creating a separate state machine for each branch, using a two-wire handshake between the branch machines and the principal machine. Goal based management is not used during this phase because decisions can be made using local graph matching and are relatively independent with little interaction. The functional goals of the derivation phase can fail only if the specification contains some unknown sequence of points or arcs. In that case an error message is printed and the design is terminated.

The second design phase is state table modification. There are two types of modifications that can be made to the state table: constraint satisfaction and optimization. Modifications currently performed by CONSPEC are for minimum and maximum constraint satisfaction and to merge parallel branches into a single state machine implementation. Parallel branches are only merged if the total number of states required to implement the fork will thereby be reduced from that of the multiple machine implementation.

All state table modifications are performed using explicit goal-based reasoning. This is necessary because the effects of modifications are not local to a single state, but span a number of states. The range of the modifications may overlap, resulting in various interactions as described in Section 4.1. All procedures that make modifications to the state table are represented as plans for the solution of explicit goals. The current state table modifications could be increased by the introduction of other techniques, either to serve as alternate plans for goals already

recognized, such as maximum time constraints, or as plans for new goals, such as loop unwinding.

The modification phase is iterative, so that after all the chosen plans have been executed, the goal detector is called again. This is necessary for three reasons. First, plans may fail. It may therefore be necessary to try more than one plan for the satisfaction of a single goal. Second, modifications made to the design may create the opportunity for other modifications to be made. New goals may therefore be detected on subsequent iterations. Third, some goal interactions are resolved by abandoning one or more goals. During the next iteration, some abandoned goals may be detected again and new plans chosen, possibly different from those chosen earlier. The iteration process ends when there are no new plans to be executed.

Backtracking to the derivation phase may occur after the modification phase has been entered. This involves a return to the DDS specification and a re-derivation of the state table with some modified parameters. For example, one plan to satisfy violated maximum time constraints calls for a reduction in the clock period. Plans for all other goals are abandoned and the single maximum time constraint plan is executed, resulting in the derivation phase being re-entered with a new, shorter clock period specification.

The number of design phases and their order are specified by a list contained in the knowledge base. CONSPEC cycles through the phases in the order given on the list. Each phase is characterized by a list of explicit goal-types. This list can be empty, as it is for the derivation phase. Goal detection, plan choice, and interaction management are performed for all explicit goals, if any, on the list, followed by the execution of chosen plans and other procedures associated with the phase. The number of phases could be increased by additions to the knowledge base, to include data path synthesis, for example. This is elaborated on further in the future research section of Chapter 6.

The goal management strategy is described in this chapter, while details of the algorithms used during state table derivation, time constraint satisfaction, and parallel branch merging are in Chapter 5.

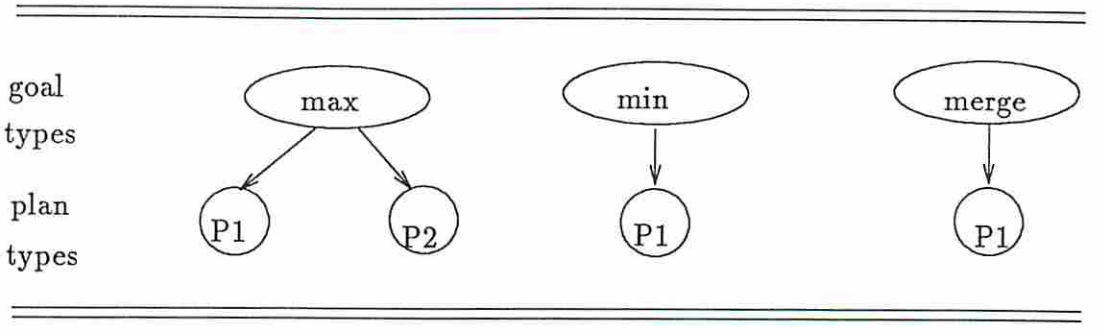


Figure 4.4: Goals and Plans in Knowledge Base

4.2.2 Knowledge Base

The knowledge base contains information on the types of explicit goals that the system can detect, solution techniques or plans for each goal type, types of goal interactions, and interaction handling techniques. There is both declarative and procedural information in the knowledge base, which is read-only during the design process. Design agents do not erase information in it, nor do they add information. Additional design information can be added to the knowledge base by the design system developer or user to increase the capacity of the system, however.

The names of known goal types are declared on a list for each phase. Associated with each goal type are one or more solution strategies or plans, each of which is a procedure in the knowledge base. The goal type list for the derivation phase is empty, indicating that there is no explicit goal-based reasoning during this phase. Figure 4.4 shows a graphical representation of the goals and plans contained in the current knowledge base for the modification phase. There are three goal types: minimum time constraint, maximum time constraint, and parallel branch merging. The maximum time constraint goal has two associated plans, while the other goals have one plan each.

There are two types of information associated with each plan in the knowledge base. The first is a precondition that is used to determine the applicability of each plan for a particular goal instance. The second is the general effect which that plan would have on the design if chosen and applied. The specific uses of this information are described in Section 4.4 on plan selection and in Section 4.5 on goal interaction management.

The types of goal interactions that the system can handle are also defined in the knowledge base. Although we refer to *goal* interactions to conform to common terminology, as Wilensky noted in [Wil83] they are more accurately viewed as *plan* interactions, since different plans for the same goal may have different impacts on the design. Interaction types are therefore identified in terms of plan effects rather than by reference to specific goals or plans. Each interaction type is associated with a meta-plan that is executed whenever an interaction of that type is detected. It is possible for different interaction types to be identified with the same meta-plan.

The current knowledge base has knowledge of ten interaction types as described in Section 4.5. Three meta-plans are defined for the resolution of goal interactions. First, particular plans or goals may be abandoned. Second, a special order may be defined on the execution sequence of the plans. Third, restricted range plans that either avoid or focus on the area of overlap between interacting goals can be chosen. The methods used to identify and manage interactions are described in detail in Section 4.5.

CONSPEC can be extended by adding new goals and associated plans or by adding new plans for the solution of existing goals. When a new plan is added, it is not necessary to add new interaction types if the plan effect is already known to the system. In that case, the addition is very simple. When a plan with unknown effect is introduced, any significant interactions between the new effect and known effects must be identified by new interaction types and associated with an interaction strategy, possibly using existing meta-plans. If these are inadequate, one or more meta-plans will need to be added for each new interaction type.

4.2.3 Blackboard

The data base is referred to as a blackboard because of its use by all design agents for writing, reading, and erasing information. CONSPEC's blackboard has four levels. Level one is the DDS Data Flow and Control and Timing Model specification; level two contains the state table derived from the specification. All goal instances are written on level three of the blackboard as they are identified by the goal detector. The interaction handler also writes meta-goals on the blackboard for some interaction types, as will be explained in Section 4.5. Level four of the

blackboard holds the plan chosen for each elementary goal; each meta-goal has only a single plan that is not written on the blackboard.

Individual design agents read from and write to particular levels of the blackboard, as illustrated by Figure 4.3 on page 85. The goal detector reads information from the DDS and state table levels and writes on the goal level. The plan proposer reads information from the goal, DDS, and state table levels and writes on the plan level. The interaction handler reads and writes on the goal and plan levels, and may also erase information on the same levels. The executor reads from the plan and state table levels and writes to the state table level.

Traditional blackboard architectures refer to the levels read from as the stimulus frame, and to the levels written on as the response frame of the agent. The term stimulus is used to indicate that the agent is stimulated to action when it detects its initiation conditions on the blackboard. Blackboard architectures were created for the needs of real time processing such as speech understanding or mobile robot collision avoidance. The real-time element means that data on the board is changing continuously and agents must be ready to act whenever appropriate. Digital design is not a real-time process, however, and we assume that the specification is not changed in the middle of the design process. For this reason the relative timing of design agent initiation can be anticipated and it is not necessary for each agent to continuously monitor the blackboard. In CONSPEC the design manager takes the place of the traditional stimulus-response model in maintaining the proper sequencing of the design agents.

4.3 Goal Detection

The types of explicit goals detectable by CONSPEC are identified by a list maintained for each design phase. Currently, no goal types are defined for the derivation phase and three are defined for the modification phase, as specified by the Prolog clauses:

```
goals(derivation, [ ]).
goals(modification, [merge, maximum, minimum]).
```

Each goal type is represented on the list by the name of a goal-procedure that can detect all instances of goals of that type. Each goal procedure returns a list

containing one argument for each goal instance found, returning an empty list if no goals of that type are found. The argument uniquely identifies the portion of the state table associated with the goal; its contents vary with the goal type.

The goal-procedure *merge* examines the DDS specification for parallel forks, delineated by points of type and and and-join,[4.2] returning as an argument a list of the start states for all branches of the parallel fork.

Both goal-procedures *minimum* and *maximum* examine the DDS specification for time constraints between DDS points. The constraints between points are converted to constraints between states by examining point-state links created during state table derivation.[4.3] The argument returned for each time constraint is a list containing the two end states and the length of the constraint. For example, the argument returned for a 50 nsec minimum time constraint between states S1 and S6 would be [[S1,S6], 50].

When the argument list is returned the goal detector establishes an explicit goal of the same type as the goal-procedure, for example minimum time constraint, for each argument on the list, provided the same argument has not been detected during a previous iteration. Because the design process is iterative, it is possible for the same goals, such as time constraints, to be detected more than once. To prevent the same goal from being asserted more than once, and also to keep track of the plans that have been attempted for goal satisfaction, whenever a goal is created a history is established in the form history(Argument,GoalName,PlanHistory). The PlanHistory is a list that is initially empty. Whenever a plan is executed for the satisfaction of this goal, its name is placed on PlanHistory. Histories are used during plan choice as described in Section 4.4.

For each argument returned by a goal-procedure, the goal detector looks for a goal history with the identical argument and goal type (goal type is easily identified from the goal name). If there is no such history, an explicit goal with the following Prolog clausal form is established, along with a new goal history:

goal(GoalName,Argument).

[4.2]Graph traversal is not required to find these points because the DDS timing graph is represented using Prolog clauses located using hashing algorithms.

[4.3]Points that signal the beginnings of states are flagged during state table derivation.

Goal names are created by appending a unique number to a root representing the goal type. The root for goal type merge is merge, for goal type minimum it is min, and for maximum it is max. Using the argument, [[S1,S6], 50], of the time constraint example given above, the goal established would be: goal(min-N, [[S1,S6], 50]), where N is a number uniquely identifying that goal, and the initial goal history would be: history([[S1,S6], 50],min-N,[ ])

An outline of the goal detection strategy follows:

1. Let Types = goal types associated with current phase
2. For each Type $\in$ Types, do:
   (a) Execute procedure Type that returns Arguments
   (b) For each Arg $\in$ Arguments, do:

       If $\nexists$ history(Arg,Type-X,Z) where X = number and Z = list
       Then create new goal and initiate history on blackboard.

4.4 Plan Selection

4.4.1 Plan Representation

Each goal type known to the system is associated with at least one plan for its solution. Available plans are specified by a list maintained for each goal type as follows: plans(GoalType,PlanList). Each plan is a procedure that is represented on PlanList by name. In the case of the goal type maximum, there are two plans, represented by the clause: plans(max,[eliminate,reduce]). Eliminate is the name of a procedure that attempts to eliminate states on long paths by asserting conditional outputs one state earlier and using clock suppression of outputs. Reduce is a procedure that queries the user to determine a new shorter clock period, followed by a re-derivation of the state table using the shorter clock period. There is one plan for the merging of parallel branches and one for the satisfaction of minimum time constraints.

A plan must be chosen for each goal established by the goal detector. Plans chosen for different goal instances of the same type can be different if there is more than one plan on the plan list of that goal type. The connections between plans

and associated goals are made explicit in the plan representation that points by name to a goal and a plan identified by procedure name, using the following clausal form:

plan(GoalName,Plan,Effect,Range)

Although currently there are relatively few goal types and associated plans in the knowledge base, our objective has been to develop a plan selection strategy that can accommodate the introduction of new goals and plans. Plan selection may be useful even when there is but a single plan available on the plan list of the goal type, because that plan may not be applicable in all circumstances. Three factors are used in CONSPEC's plan selection strategy: the order of appearance of a plan on the plan list, the history of plan choice for each goal, and the precondition associated with each plan.

4.4.2 Order of Plans on Plan-List

Each time a plan is chosen for a goal, the corresponding goal type's plan list is examined from the first plan to the last. This is true even if previous plans chosen for the same goal have failed and plan choice is being performed again. The first plan found to be applicable is chosen; the next two sections on preconditions and plan histories define plan applicability.

The plans may be ranked using various criteria, depending on the goal type; plans may also be listed in arbitrary order. Chapter 5 includes a description of two plans available for maximum time constraint satisfaction; these are ranked on the plan list by their effect on state machine size. The first plan on the list eliminates states where possible on paths that are too long, thereby reducing the number of control states. The second plan on the list consults with the user to shorten the clock period; this may reduce processing time but also increase the number of control states. Other measures such as performance or design time could be used to rank plans for goal types not included in the present implementation of CONSPEC.

4.4.3 Plan Histories

If a chosen plan succeeds during plan execution, its associated goal also succeeds. If the plan fails, however, on the next iteration the plan proposer will attempt to find another applicable plan from the goal type's plan list, again starting from the first plan on the list. If a plan has already been chosen for the same goal on a previous iteration, however, it is passed over and the next plan on the list is examined.

CONSPEC keeps track of the plans chosen for each goal using the PlanHistory list introduced in Section 4.3. During goal detection a history list is established for each goal; whenever a plan is executed the plan name is placed on the corresponding goal's history list. The existence of a plan on the history list indicates that it has already been tried and failed on some previous iteration. For this reason it will be rejected and the next plan on the plan list considered.

It may be possible that changes made by a plan A are necessary for the successful execution of some other plan B. If plan B is executed prior to plan A, therefore, it will fail. Because plan B is now on the history list for that goal, however, it will not be chosen and executed again, even after plan A executes. These situations are anticipated by meta-plans that specify an order on the execution of interacting plans or (temporarily) abandon a single interacting goal. Thus if the interaction between plan A and plan B is represented by the appropriate interaction type, either plan A can be executed before plan B, or the goal associated with plan A can be abandoned and reinstated in a later iteration. If the goal associated with plan B is detected at an earlier iteration than that for plan A, however, this cannot be resolved so easily. For the present system goals and plans this situation will not arise. In the general case it is important to avoid retrying the same plans over and over. It may be possible, however, to override the history list proscription by searching for particular interaction types between failed plans and plans executed during later iterations.

4.4.4 Plan Preconditions

It was mentioned in Section 4.2.2 that each plan in the knowledge base is associated with a precondition that is used during plan choice. Preconditions are procedures

that examine information on the blackboard and return a value *true* or *false* when run with a goal argument as a parameter; a precondition that is always *true* can also be used. The precondition is designed to screen out plans that are obviously inapplicable and have no chance of success. A plan is therefore considered to be applicable if first, it is not on the goal's plan history list and second, if its precondition is true. The first plan on the plan list that meets these two criteria is chosen; if there are no more applicable plans the goal fails.

Because of preconditions the order of plan choice can be different from the order in which they appear on the plan list. A precondition that is false during one iteration of the design cycle may, owing to changes to the state table made by other plans (the merging of parallel machines, for example), test true during a subsequent iteration. This is why plan choice must always begin with the first plan on the goal type's plan list. The combined effect of preconditions, plan ordering, and plan histories accomodates plan choice strategies of varying complexity, ranging from the trial of one plan after another in simple sequence to an order of plan choice that depends on specification characteristics determined by plan preconditions.

In the current version of CONSPEC preconditions are important in the plan choice for only a single goal-type, maximum time constraint. The precondition for the first plan on the list is a procedure that always returns true. This plan is therefore always chosen during the first iteration. If this plan is not successful at satisfying the time constraint, during the next iteration the plan proposer passes over the first plan, since it is on the history list, and tests the precondition for the second plan. This precondition is a procedure that consults the user on the advisability of reducing the clock period. The user is provided information on the current clock period and a list of interval lengths from the specification. If the user refuses to shorten the clock period, the precondition fails and, because there are no more plans available for the satisfaction of maximum time constraints, the goal fails also. If the user suggests a new clock period, the precondition succeeds and the plan is chosen, resulting in backtracking to the derivation phase. Note that the reduction of the clock period is not guaranteed to meet the maximum time constraint, however, so the constraint goal may yet fail.

The precondition of the single plan for goal type merge parallel branches always returns true, although that does not necessarily mean that the merge will

be performed. That depends on the number of states in the merged machine versus the separate machines, as described in Section 5.3.2. The number of states and next state transitions of the merged machine will explode unless the branch machines are simple and there are no internal loops. The plan therefore fails if the parallel fork cannot be implemented in fewer states using a single machine. The precondition for the single plan for goal type minimum time constraint is also always true. In the absence of negative interactions, explained in more detail in Section 4.5, this plan invariably succeeds since it is always possible to add delay to the execution time.

4.4.5 Plan Selection Algorithm

To summarize, the plan proposer takes the following steps to chose a plan for each goal:

1. Let Type-N = goal instance, where Type is the goal type.
   Let PlanList = list of plans available for Type
   Let History = record of plans previously chosen for Type-N
   Let i = 1

2. Let precond-Plan = ith element of PlanList

3. If (no ith element) Then (Type-N = failed).
   Elseif (Plan $\in$ History) Then (i = i + 1, Go to 2).
   Elseif (precond $\rightarrow$ false) Then (i = i + 1, Go to 2).
   Else assert plan(Type-N,Plan,Effect,Range) on blackboard

As mentioned in Section 4.2.2, every plan has an associated effect that represents the type of impact the plan has on the design. The range is a measure of the extent of the plan's impact. Both are used to detect interactions between goals and are discussed in detail in the next section.

If no plans on the list are found to be applicable, the goal necessarily fails. The significance of goal failure depends on the type of goal involved. The failure of a constraint goal indicates a design failure and must therefore be reported to the user, but the failure of an optimization goal, such as the merging of parallel branches into a single machine implementation, need not be reported to the user.

All goal types that represent constraints are denoted in the knowledge base by a clause of the form: constraint(GoalType). If a constraint goal fails, the plan proposer removes the goal, but not its history, from the blackboard and places it on a fail list for later analysis. Otherwise the goal is removed but is not placed on a fail list. Failure analysis is discussed in Section 4.7.

4.5 Interaction Management

4.5.1 Representation of Interactions

A primary motivation for the use of explicit goal-based reasoning is the need to resolve interactions between individual design decisions. Figure 4.1 on page 80 was used to illustrate the kinds of interactions that can occur during modification of the state table. The interaction handler uses general information to detect potential interactions; the actual significance will depend on characteristics of the particular specification.

A traditional planning approach to detecting interactions uses operator preconditions and postconditions. For example, if the postcondition of an operator is to remove some object whose existence is the precondition for another, the interaction becomes apparent. Taking loop unwinding as an example, the precondition could be: loop exists and the number of iterations is constant, with the postcondition: loop removed. For the merging of parallel branches, the precondition is: loop doesn't exist. A traditional planner could detect this interaction and order the plans so that the loop is unwound first.

For some interactions, however, the pre and postcondition technique is not suitable. For example, the satisfaction of time constraints is simplified if loops within the range of the constraint can be unwound first. It is not always desirable to unwind loops, however, particularly long ones, even if the number of iterations is a constant. Preconditions represent conditions that *must* be true before applying a plan; for this reason, we do not want to specify the absence of loops as a precondition for applying the time constraint plans. It is also awkward to use pre and postconditions to detect the positive interaction between nested minimum time constraints, since it requires a direct reference to other goal types. An additional

complication is that plans only interact if the ranges of their actions overlap, that is, if the same state sequences are affected.

Instead of using pre and postconditions, we associate two measures with each plan that are used to detect goal interactions: ranges and effects. The effect represents the general impact that execution of a plan will have on the state diagram while the range measures the extent of the impact. We first define the term range and describe its importance, before going on to a discussion of plan effects.

The range of a plan is the part of the graph that may be modified in some way when the plan is executed. CONSPEC considers interactions only between plans that affect the same part of the design or have nested or overlapping ranges. In the example of Figure 4.1 on page 80, a plan to merge the outer parallel branches would have a range extending from *and1* to *and_j1*. This range is represented by the end points [and1,and_j1], and includes all points on all branches between these two points. All three time constraints and the parallel fork at *and1* (range [and2,and_j2]) are nested within the range of the parallel fork at *and2*. The ranges of the two minimum time constraints are nested within the maximum constraint, but do not overlap with that of the parallel fork at and2.

A procedure is run for each goal to determine the range of a chosen plan's effect over the design; the range of a single plan can encompass the whole design if its impact is global, or only a local area. A traversal through the DDS graph is used to determine the relation between the ranges of different plans. The DDS timing graph is used instead of the state table because it is connected, unlike the state machine implementation, which may comprise multiple state tables that implement parallel branches. The connection between the DDS and the state diagram is represented by links created during state table derivation between each state and the DDS point that marks its beginning.

The second measure used in detecting goal interactions is plan effect. Effects are associated with plans rather than goals because it is possible for the different plans of a single goal type to have different impacts on the design. Four effects are currently defined: add, remove, backtrack, and join. The minimum time constraint plan adds timing states to short paths and therefore has the effect add. The first plan for maximum time constraint satisfaction eliminates states by asserting conditional outputs early and so has the effect remove. Both add

and remove refer to modifications on sequences of states. The second maximum time constraint plan has the effect backtrack, since it reduces the clock period and restarts the design process at the derivation phase. The merge parallel branches plan has the effect join.

4.5.2 Detection of Interactions

Interactions are distinguished between two plans at a time, defined in terms of plan effects. A combination of effects taken two at a time is called an interaction type. The interaction between two plans with effects *reduce* and *add*, for example, is identified by the interaction type reduce-add, or R-A. With four effects defined there are a total of ten possible interaction types.

Every interaction type that may have a significant impact on the design, either positive or negative, is associated with an interaction strategy; some interaction types have identical interaction strategies. When an interaction of a particular type is detected among the plans on the blackboard the corresponding interaction strategy is executed. Various procedures termed meta-plans assert meta-goals and modify goals and plans on the blackboard, and are used in carrying out the interaction strategies. Three meta-plans, Abandon Goal, Order Execution and Restrict Range are currently implemented. Different interaction strategies may share the same meta-plan; for certain interaction types more than one meta-plan must be executed.

Figure 4.5 illustrates the relations between goals-types, plans, effects, interaction types, and meta-plans for the modification phase. Effects are referred to by the first letter, R for reduce, B for backtrack, A for add, and J for join. Interaction types can be defined using variables to refer to one or both effects; this is done when different interaction types share the same interaction strategy. For example, the four interaction types B-B, B-R, B-A, and B-J have identical interaction strategies and therefore are represented by B-X, where X is a variable representing any effect. Similarly, Y in the figure is a variable that can take on the value R or A; Y-Y stands for the interaction types R-R and A-A, and J-Y stands for the interaction types J-R and J-A.

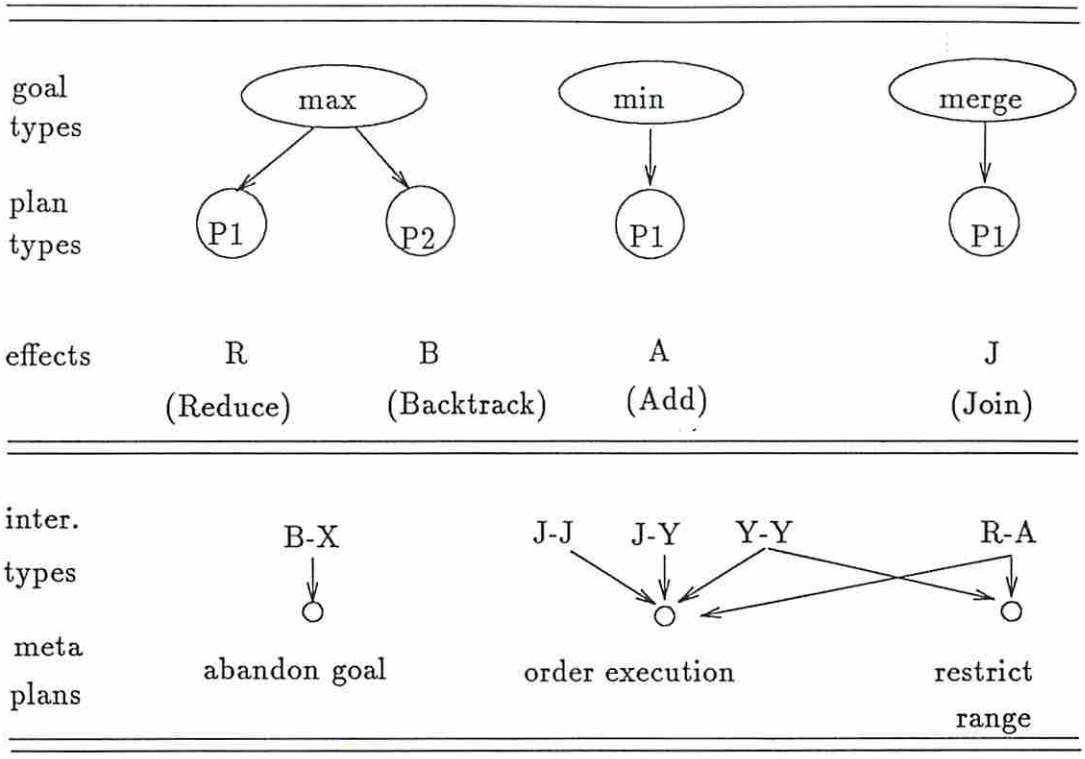


Figure 4.5: Representation of Interactions

In the succeeding paragraphs we outline the actions and results of the three meta-plans. Following that, the strategies used by the interaction handler for the ten interaction types are discussed in the same order that they appear in Figure 4.5.

4.5.3 Metaplan Actions and Results

The meta-plan **Abandon Goal** manages a conflict between two goals by removing one goal along with its associated plan and plan history from the blackboard. By abandoning one goal we eliminate the conflict and permit the other goal's plan to execute. In some cases the abandoned goal will be re-detected during the next iteration by the goal detector; the plan chosen may be different from the one chosen during the earlier iteration, however. In other cases the goal may be permanently abandoned. Interactions that potentially lead to goal abandonment are identified before any other interaction types. The Abandon Goal meta-plan is executed as soon as the interaction is detected, and prior to the execution of any other type of

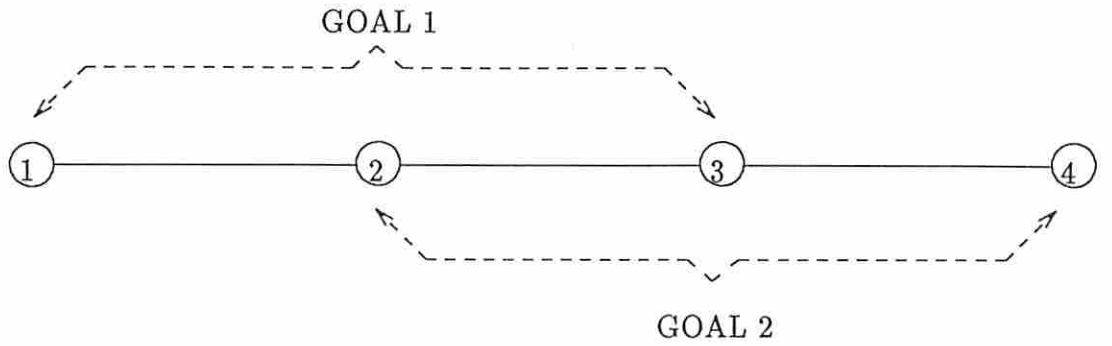


Figure 4.6: Overlapping Ranges

meta-plan. In this way time is not wasted processing interactions involving goals that may be finally abandoned.

The meta-plan **Order Execution** manages an interaction between two goals by specifying an order of execution on their plans. To specify that the plan for GoalX is to executed before that of GoalY, for example, the clause **order(GoalX,GoalY)** is asserted on the blackboard. There may be many interactions detected during a single design iteration that result in the assertion of **order** clauses, and a particular goal may be referred to in more than one of these clauses. The collection of all order clauses defines a partial ordering among the plans that specifies their proper execution sequence. This partial ordering is determined by the executor prior to plan execution, in a manner described in Section 4.6.

The meta-plan **Restrict Range** restricts the range of plan effects. We refer to Figure 4.6 to illustrate the technique. This figure shows a schematic representation of the overlapping ranges of two goals. If the plan effects of Goal 1 can be limited to that part of the range that does not overlap with Goal 2's range, between points 1 and 2, then the plan execution will not interfere with Goal 2. In this way negative interactions can be avoided. Conversely, we can restrict the plan effects of Goal 1 to the overlapping range from point 2 to 3. This allows us to take advantage of a positive interaction between two goals whose plans have the same effects.

The meta-plan Restrict Range is implemented by finding a plan for goal satisfaction that accepts a restricted range. The plan list of the goal type is examined for a plan whose precondition is a procedure named **restrict**.[4.4] The original plan is removed from the blackboard and replaced with the new plan. If range restrictions

[4.4]These plans exist for goal types maximum and minimum, but are not included in Figure 4.5.

are required for both goals involved in an interaction, the meta-plan is executed for each. If the goal cannot be satisfied within the range restriction, it can be lifted and the plan allowed to execute within the entire range. The consequences of this are addressed within the discussion of interaction strategies Y-Y and R-A, which employ the Restrict Range meta-plan.

4.5.4 Interaction Strategies

Now that the three meta-plans have been described, we move on to discuss five interaction strategies. The first, represented in Figure 4.5 by B-X, applies to the four interaction types B-J, B-R, B-A, and B-B. Backtracking indicates that design decisions already made are to be undone and the design restarted with some parameters changed. For example, the second plan for the satisfaction of maximum time constraints abandons the state table already created and returns to the derivation phase with a shortened clock period. The effects of executing a backtracking plan are global, therefore by definition its range covers the entire design. For this reason, a backtrack plan overlaps with and interacts with all other plans on the blackboard.

Because design decisions made before the backtrack plan is executed are undone, there is no point in executing new plans. For this reason, the action taken in response to each interaction of type B-X is to call the meta-plan Abandon Goal and remove the plan with effect represented by X. The total effect of these interactions is the removal of all goals other than the backtracking one.[4.5] B-X interaction types are handled before any others to avoid wasting time on goals and plans that are ultimately abandoned.

Interaction type J-J is between two plans with the effect join. The plan to merge parallel branches into a single machine implementation has the effect join. Because CONSPEC specifications contain only properly nested branching constructs, the only way two plans with this effect can interact is if the range of one plan is nested inside the other. As discussed with reference to Figure 4.1, branches of parallel forks can never be merged into a single state machine implementation unless the branches of any nested parallel forks have already been successfully merged. For

[4.5] If there are multiple plans with effect backtrack, all but one are abandoned.

this reason, we wish to delay a merge decision on the outer parallel fork until the inner merge plan has been executed. This is done by calling the meta-plan Order Execution to execute the nested merge plan before the outer one. If the nested parallel branches can be merged, then it may be possible to merge the outer parallel branches as well.

The third interaction strategy, represented in Figure 4.5 by J-Y, applies to interaction types *J-R* and *J-A*, between a plan with effect join and another with effect reduce or add. Plans with effect add or reduce affect sequences of states by increasing the number of states or eliminating states, respectively. Plans with effect join affect states across parallel branches by joining them. One heuristic we use, explained earlier in conjunction with the example in Figure 4.1 on page 80, is the execution of plans affecting single branches prior to those acting across branches. For interaction type J-R, it is harder to reduce the delay between two events on a branch after a merger between that branch and others of a parallel fork. In the case of interaction type J-A, the addition of delay between two events on a branch will have a greater impact on performance if done after a merger. Both interaction types are handled by calling the meta-plan Order Execution to execute the plan with effect join after the other plan.

Interaction types R-R and A-A, represented in Figure 4.5 by Y-Y, are positive interactions. The interacting plans have the same effect and can combine their actions in a mutually beneficial way. Referring to the overlapping goals in Figure 4.6, the effects of the two goals, either the addition or removal of states, could be focused in the region of range overlap between points 2 and 3. In this way the satisfaction of one goal will contribute to the satisfaction of the other goal. To accomplish this objective, for each such interaction the meta-plan Restrict Range is called. If the goal cannot be satisfied within the restricted range, the restriction is automatically lifted. This can be done without negative impact on the design because the interaction is positive.

When the interacting plans have nested ranges, as shown in Figure 4.7, we can take advantage of the positive interaction in a simpler way, by executing the nested plan first. In this way the effects of the nested plan count towards satisfaction of the outer goal. The meta-plan Order Execution is called to execute the nested

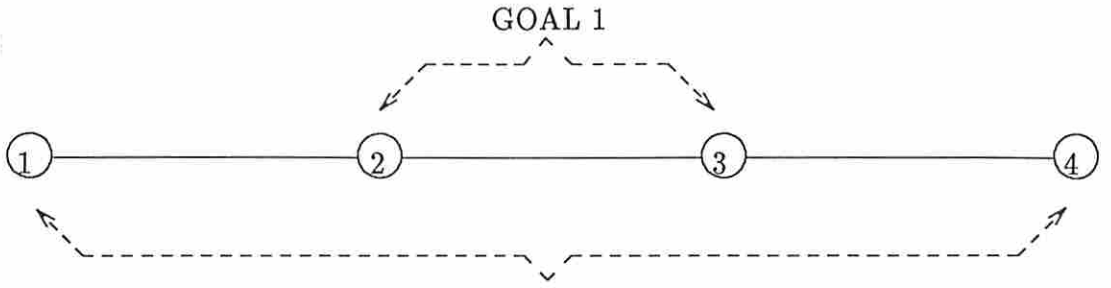


Figure 4.7: Nested Ranges

goal, Goal 1 in the figure, prior to the outer goal, Goal 2. No range restriction is required in this case.

The last interaction type, R-A, is difficult to resolve since the conflicting plans are counteractive: one plan's effect is to remove states, while the other's is to add states. To be most effective the interaction strategy must take into account the relatively greater difficulty in reducing as opposed to increasing the number of states.

We consider four different cases of this interaction type based on the relations between the conflicting plans' ranges. The first case is that of overlapping ranges, shown in Figure 4.6 on page 101, where Goal 1's plan (Plan 1) has effect *add*, and Goal 2's plan (Plan 2) has effect *remove*. For this case, which corresponds to overlapping minimum and maximum time constraints, we use two meta-plans to control the interaction. First, Plan 1's effect (i.e., the addition of states) is restricted to that part of its range, between points 1 and 2, which does not overlap with Plan 2's range. This is done by calling meta-plan Restrict Range. We do not place a similar restriction on Plan 2. Instead, the execution of Plan 2 is scheduled to take place before Plan 1 by calling meta-plan Order Execution. If Plan 2 removes any states between points 2 and 3, Plan 1 will then be able to compensate by the addition of extra states between points 1 and 2. If Plan 2 fails, it will not be due to any interaction effects, because of the restricted range imposed on Plan 1. Plan 1 will not fail, because it is always possible to add states.

The second case of interaction type R-A is for plans with identical ranges, as shown in Figure 4.8. This corresponds to the specification: the time between points

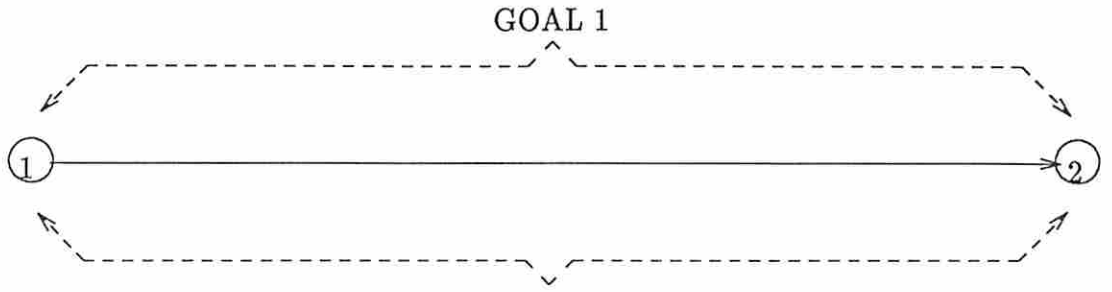


Figure 4.8: Identical Ranges

1 and 2 should be greater than or equal to t_{min}, and less than or equal to t_{max}.[4.6] Provided that the value t_{min} is less than or equal to t_{max}, one of the goals must be trivially achieved (that is, already satisfied). If the delay from point 1 to point 2 exceeds the maximum and must be reduced, it will also exceed the minimum; if it is less than the minimum and must be increased, it will also be less than the maximum; otherwise the delay must be within the acceptable range between the minimum and the maximum. If $t_{max} < t_{min}$ the two goals are incompatible; an error message is printed and the user is queried for new values. Otherwise, the plan of the one goal that is not satisfied (if any) is executed.

The next two cases of interaction type R-A are for plans with nested ranges, as shown in Figure 4.7. First we consider the case where Goal 1's plan (Plan 1) has effect *remove*, while Goal 2's plan (Plan 2) has effect *add*. This corresponds to a local maximum time constraint nested within a global minimum time constraint. The interaction strategy is virtually identical to the case of the overlapping plans, and for the same reasons. The nested Plan 1 is executed without a range restriction prior to Plan 2. Plan 2 is then executed with its range restricted to avoid the area between points 2 and 3 (Plan 1's range). Again, two meta-plans, Order Execution and Restrict Range, are used. If the nested goal cannot be satisfied, it will not be due to any interaction with the outer goal (owing to the range restriction). Goal 2 can always be satisfied, regardless of the effect of Goal 1, because extra states can always be added.

[4.6] An exact value, where $t_{min} = t_{max}$, cannot be achieved unless it is a multiple of the control clock.

The last case we consider is the most difficult to handle. Again, the ranges are nested, as shown in Figure 4.7, but this time the plan effects are reversed: Goal 1's plan (Plan 1) has effect *add*, while Goal 2's plan (Plan 2) has effect *remove*. This corresponds to a local minimum time constraint, Goal 1, nested within a global maximum time constraint, Goal 2. The meta-plans employed to treat the interaction are identical to the previous case. The nested plan is executed before the outer plan by calling Order Execution. Meta-plan Restrict Range is called to impose a range restriction on the outer plan, avoiding the range of the nested plan. This way Plan 2 can attempt to compensate for the effects of Plan 1, while avoiding its range. The effects of the two meta-plans can be summarized by the following clauses placed on the blackboard:

order(Goal1,Goal2)
restrict(Goal2,Goal1)

In addition, the interaction handler determines if any path between points 2 and 3 meets the constraint of Goal 1. If not, the execution of Plan 1 will result in the lengthening of the critical path with a possible negative effect on satisfaction of the outer goal. In that case the interaction handler places the meta-goal restrict(Goal1,Goal2), on an interaction fail list. This clause signifies that the effects of Plan 1 could not be restricted to avoid the range of Plan 2, and that the satisfaction of Goal 1 may have a negative effect on Goal 2. If at least one path does meet the constraint, however, Plan 1 will not lengthen the critical path and consequently the satisfaction of Goal 1 does not have a negative effect on Goal 2.

The difficulty with an interaction of this type is that Plan 2 can fail if it is unable to remove enough states within a restricted range to satisfy Goal 2. Recall that it is always more difficult to reduce the number of states than it is to add extra ones, and may not always be possible. In that case we have two options: first, we can report failure of the outer goal, or second, we can lift the restriction and allow the outer plan to act over its full range. The choice depends on whether it might be feasible to satisfy the outer goal without violating the nested goal by lifting the restriction.

If the minimum time constraint requires a lengthening of the critical path between points 2 and 3, then it will be violated if we satisfy the maximum time constraint by lifting the range restriction and shortening the same paths. In that

case, which is indicated by the presence of the clause restrict(Goal1,Goal2) on the interaction fail list, we choose the first option and report failure of the maximum time constraint.

Conversely, if the minimum time constraint was shorter than the critical path between points 2 and 3, then it may be possible to satisfy the maximum time constraint by lifting the range restriction without violating the minimum time constraint. In that case, we choose the second option, place the clause restrict(Goal2, Goal1) on the interaction fail list, and try to satisfy the maximum constraint without the range restriction. This action may result in the violation of the nested goal and therefore must be monitored; this is discussed in Section 4.7 on failure analysis.

In general, during execution of a restricted range plan for any goal *GoalY*, where the meta-goal restrict(GoalY,GoalX) defines the range restriction, in case of failure the restriction is lifted only if there is not another meta-goal restrict(GoalX,GoalY) on the fail list. The rationale for this is that there is no hope of success in the presence of mutual interference between two negatively interacting goals.

4.5.5 Analysis of the Interaction Strategy

The interaction management techniques described above deal with interactions between two goals at a time. In the case of all but one interaction type, the methods described will also handle multiple (more than two) mutually interacting goals. This section provides an analysis of the performance of the interaction strategy described in the previous section.

In the case of interaction type B-X (interaction of a plan with effect backtrack and a plan with any other effect), the meta-plan *abandon goal* is applied sequentially to all interacting plans. Because a backtrack plan has global range, it interacts with all other plans; the combined effect is removal of all plans besides the backtrack plan. If there is more than one plan proposed with effect backtrack, then only one of these plans will actually be carried out. The current version of CONSPEC has a single plan type with effect backtrack, so the execution of multiple *instances* of the backtrack plan (to change the clock period) would not be useful anyway. If different backtrack plan *types* existed, however, it would be

necessary to define a separate B-B interaction type along with a new meta plan to combine the effects of the different plans.

Several interaction types use the meta plan *order execution.* These include J-J (two plans with effect join), J-Y (plan with effect join plus another plan with effect add or reduce), nested(Y,Y) (two plans with the same effect, either add or reduce), and R-A (plans with effects reduce and add). Although specified between plans two at a time, the *order execution* meta-plan can handle any number of plans with nested or overlapping ranges, in any combination of interaction types. This is because the plan executor is able to determine the proper sequence of multiple plans constrained by order clauses.

The collection of ordering clauses specifies a partial order on goal solution. With the current interaction types, it is always possible to achieve all ordering relations, that is, there will never be an order relation that cannot be achieved because it is cyclic (e.g., A precedes B that precedes C that precedes A). Cyclic order relations can result only if some goal appears as first and second argument in different order clauses (a necessary but not sufficient condition). Goals whose plans have effect **join** only appear as second arguments and never as first (interaction types J-A and J-R). Goals whose plans have effect **reduce** or **add** can appear as first argument, when nested, and second argument, when enclosing a nested goal (interaction types nested(X-X) and nested(R-A)). It is impossible to have a cyclic ordering relation among goals defined in terms of nested ranges, since the nesting relation is transitive, that is, if A is nested inside B and B within C, then A is also nested within C and never the other way around. Goals whose plans have effect **reduce** can also appear as first argument when overlapping with another goal whose plan's effect is **add**. These clauses also cannot create cyclic orderings because the goal whose plan has effect **reduce** is always the first argument and never the second.

Interaction type R-A is a negative interaction between plans with effects reduce and add. The interaction strategy uses a combination of the meta-plans *order execution* and *restrict range.* For any two interacting goals, if a solution exists by adding and deleting states, this method will find it. It is obvious that for overlapping ranges or when plan(R) is nested within plan(A), no matter how many states are removed to satisfy plan(R) within the range of plan(A), plan(A) can

compensate by adding states within its non-overlapping range. For these cases, therefore, if a solution exists, it can be found.

When plan(A) is nested within plan(R) two possible solution types exist, either of which the interaction strategy can find. First, assume that it is possible to remove states from the non-overlapping range of plan(R) to meet the outer constraint, taking into account any addition of states by the nested plan(A). Since CONSPEC schedules plan(A) first and restricts plan(R) to avoid plan(A)'s range, this type of solution can obviously be found. Second, assume that a solution requires removal of some states from within the range of plan(A). The number of states removed within that range of overlap should be sufficient to meet the outer constraint without violating the inner constraint. The interaction strategy finds this solution by removing as many states as possible from the non-overlapping range of plan(R), and the minimum number of states from within plan(A)'s range. Plan(R) is first executed with a restriction to avoid plan(A)'s range, so as many states as possible are removed from the non-overlapping range. When this fails to remove enough states to satisfy the constraint, the range restriction is lifted[4.7]. This means that only as many states as are needed to meet the constraint are removed from within plan(A)'s range. For this reason, it cannot happen that states are removed from within plan(A)'s range when others outside the range should have been removed instead.

If there are multiple mutually interacting plans with effects reduce and add, and if a solution exists such that all negative interactions can be avoided or compensated for within non-overlapping regions, then the interaction strategy can find the solution.[4.8]. In the general case, however, the interaction strategy described is not guaranteed to find a solution for this interaction type when multiple goals are involved. This is because more information is required to determine where states should be added or removed on the multiple range segments defined by the goal overlaps. As a result, although all minimum constraint goals can be satisfied, existing solutions for maximum constraint goals may not be found.

[4.7]Provided that avoid-range(plan(A),plan(R)) is not on the fail list, as described in the previous section.

[4.8]The implementation of the meta-plan *restrict-range* could be extended without any fundamental difficulty to accept multiple range restrictions.

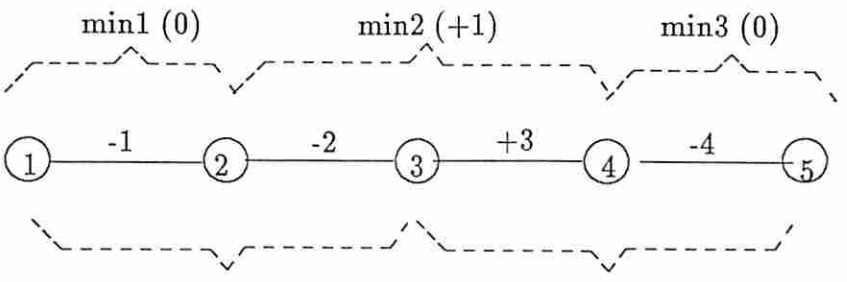


Figure 4.9: Multiple Conflicting Goals

The example in Figure 4.9 shows the ranges of five overlapping goals (the actual state transitions are omitted). Max1 requires the elimination of three states for its satisfaction; max2 requires the eliminatiion of one state; the critical path through min1's range exceeds the constraint by one state; min2's critical path requires one timing state; and the critical path through min3 exceeds the constraint by four states. Let us assume that the following solution exists (as shown by the numbers on the figure): remove one state between points 1 and 2, two states between points 2 and 3, and four states between 4 and 5, and add three states between points 3 and 4. The *restrict range* meta plan will not work for this example, since the ranges of all goals overlap at every point with some other conflicting goal.

It is unlikely that multiple conflicting goals of this complexity will frequently occur in specifications. To deal with this situation, however, we can formulate it as a general Constraint Satisfaction Problem (CSP) [Win84, Dec86] and use a dependency-generated backtracking approach to find a solution. This method uses local constraint propagation and search techniques to achieve global consistency; it is appropriate and useful for the case of mutual multiple interactions of type R-A, but is not required for other interactions dealt with in this thesis. The technique, which has not been implemented in CONSPEC, could be incorporated into the existing system as a meta-plan. The interaction strategy would determine a list of mutually interacting goals, remove the plans for each individual goal, and substitute instead a single CSP meta-plan, formulated as described below.

The problem is to find which subrange of a plan to add to or reduce the number of states, and how many states to add or subtract within each subrange. Our Constraint Satisfaction Problem can therefore be expresssed using a set of n variables, $R_1, \ldots, R_n$, and a set of constraints. Each variable represents a subrange

delineated at each end by the range of some plan, where n is the total number of subranges. For our example, there would be four variables corresponding to the four subranges as follows: R_1, points 1 to 2, R_2, points 2 to 3, R_3, points 3 to 4, and R_4, points 4 to 5. The values of the variables represent the number of states to be added to or removed from the corresponding subrange, with negative values indicating removal, and positive values indicating addition.

The constraint equations indicate which values of the variables are compatible with each other, there being one constraint equation for each interacting plan. For our example, the critical path of min1 exceeds the constraint by one state. To avoid violating min1, therefore, no more than one state can be removed from min1's range (any number can be added). This gives us the constraint: $R_1 \geq -1$. Min2 requires the addition of one timing state. Its range consists of two subranges represented by R_2 and R_3, which gives us the constraint: $R_2 + R_3 \geq 1$. Similarly, the other three plans result in the following constraints:

$$\begin{aligned} R_1 + R_2 &\leq -3, \\ R_3 + R_4 &\leq -1, \\ R_4 &\geq -4 \end{aligned}$$

The constraint equations will not generally be sufficient to determine a unique solution using constraint propagation, so in addition search is required to find the final solution. This CSP meta-plan has not been implemented.

4.6 Plan Execution

During each iteration of the modification phase the executor examines the blackboard following plan proposal and interaction handling, and executes any plans present there. Some plans may be constrained in their execution sequence by clauses of the form: *order(Goal1,Goal2)*. The executor runs a procedure that places constrained goals into the required order on a list called Ordered-Goals. Plans of goals on this list are executed in the specified order, and prior to the execution of plans for goals not constrained by ordering clauses.

The Ordered-Goals list is created one element at a time, starting with the earliest scheduled goal. A goal is added to the list as soon as all goals that must precede it have already been scheduled. Only goals that are constrained to be

executed before some other goal, that is, appear as the first argument *Goal1* of some clause *order(Goal1,Goal2)*, are placed on the list. The algorithm that creates the Ordered-Goals list is outlined below. Ordering relations are traversed like a linked list to schedule each goal.

1. *initialize List that will contain scheduled goals*
   Let List = [ ]

2. *Find some goal constrained by an order relation that has not been scheduled yet, initialize variable PlaceNext with this goal*
   If ($\exists$ order(GoalX,GoalY) s.t. (GoalX $\notin$ Ordered-Goals))
   Then (PlaceNext $\leftarrow$ GoalX)
   Else (Ordered-Goals $\leftarrow$ List) *If none, scheduling finished*

3. *Find any unscheduled goal that must precede PlaceNext*
   If ($\exists$ order(Goal,PlaceNext) where Goal $\notin$ List)
   Then (PlaceNext $\leftarrow$ Goal, Go to 3) *Found new PlaceNext*
   Else (Put PlaceNext on List, Go to 2) *Schedule PlaceNext*

When the goal ordering has been determined, the plans on the list are executed, followed by those not on the list. As each plan is executed it is placed on the plan history list maintained for the goal. If the plan succeeds, the goal succeeds, in which case the executor removes the representation of the goal, but not its history, from the blackboard. If the plan fails, the goal may yet be achieved by executing some alternate plan. For example, the first maximum time constraint plan attempts to reduce the number of states on long paths by asserting conditional outputs one state earlier. If this plan fails to satisfy the constraint, one more is available that reduces the control clock period and rederives the state table. Because there may be more than one plan available for goal satisfaction, therefore, goal failure occurs only if the Plan Proposer cannot find an applicable plan.

4.7 Failure Analysis

When the plan selection process adds no plans to the blackboard for execution, then failure analysis is performed. There are two fail lists that are examined, one

for failed constraint goals and the other for failed interaction or meta-goals. The failed meta-goals are only significant so far as they impact constraint goals, first to indicate possible reasons for failure and second to signal likely goal violations that have not been detected.

Recall that if a plan with a range restriction fails, the attempt to avoid the negative interaction is abandoned under certain circumstances and the restrict-range goal is placed on the fail list. Each failed meta-goal, restrict(GoalX,GoalY), indicates that GoalX has trespassed on GoalY's range. If GoalY is known to have failed (is on the goal fail list), the failure analyzer reports the negative interaction to the user as a possible contributing factor in the failure. If GoalY is not on the fail list, the failure analyzer checks on the order of execution of the two goals. If the clause order(GoalX,GoalY) is on the blackboard, this indicates that GoalY was satisfied after GoalX. Since PlanY succeeded in satisfying GoalY in spite of the negative effects of PlanX, no further action is necessary.

If order(GoalX,GoalY) is not on the blackboard, GoalY may have been satisfied before GoalX and inadvertently violated when PlanX was executed. Under these circumstances, therefore, GoalY must be verified. Verification is performed by a plan available for that purpose on the plan list of the goal type.[4.9] We identify such a plan by its precondition *verify*. This plan is executed, returning true if the goal has been satisfied, and false if the goal has been violated. If false, the interaction with GoalX is reported to the user as the reason for failure of GoalY.

Any failed constraint goals that have not been reported to the user in the process of examining the failed meta-goals are relayed to the user as well, using error messages and providing information specific to the goal type. For goal type maximum, the amount of time by which the goal failed is given.

4.8 Summary

The following is an outline of the goal-based management strategy:

1. Goal Detection–for each goal type find all goals.

2. Plan Proposal–for each goal detected:

[4.9]The goal type *minimum* has a verification plan defined that is not shown in Figure 4.5.

(a) Chose one plan

(b) If no more applicable plans, goal = failed

3. Interaction Handling–for each interaction type:

(a) Find all interacting pairs of plans

(b) For each pair: execute meta-plan

4. Plan execution

(a) If no more plans on blackboard, go to 5

(b) Order plans as constrained by order clauses

(c) For each plan on blackboard, do:

i. Execute plan and place on history list of associated goal

ii. If (plan succeeds) Then (remove goal from blackboard)

iii. Go to 1.

5. Failure Analysis

(a) For all failed *restrict-range* goals:

i. Verify any goal not on fail list whose range was violated.

ii. If verification fails, goal = failed.

(b) Report all failed goals, including association with any failed *restrict-range* goal.

In the next chapter we describe the algorithm that is used to derive the state table from a DDS Timing and Control graph, as well as the four procedures that function as plans during the modification phase of design to satisfy minimum and maximum time constraints and to merge parallel branches into a single state machine implementation.

Chapter 5

Basic Controller Synthesis Techniques

In Chapter 3 we described the first steps leading to synthesis: specification of the desired behavior using a hardware descriptive language, SLIDE, and the internal representation of the behavior using the DDS. In Chapter 4 we described CONSPEC'S architecture and management of the design process using goal-based reasoning. In this chapter we describe the algorithms that provide the foundation for control synthesis by CONSPEC.

Because the controller governs data path behavior it is necessary to complete data path synthesis before the design of the controller. The general approach described in Chapter 1 is to perform data path synthesis from the information contained in the Data Flow Model, and control synthesis from the Control and Timing Model. The first section of this chapter outlines an approach to data path synthesis using existing programs in the USC ADAM system.

Figure 4.2 on page 85 in Chapter 4 shows the two basic steps used by CONSPEC in arriving at an implementation for the controller. During the first step, called the derivation phase, a state table specification of a Finite State Machine is derived from the Timing and Control graph. As described in Section 1.4, the Mealy model is used, in which controller outputs may depend on the values of controller inputs as well as state. Section 5.2 of this chapter describes the procedure executed during the derivation phase of design.

During the second step of the design, called the modification phase, modifications may be made to the state table to optimize the design or meet user specified timing constraints. As described in Chapter 4, explicit goal-based reasoning is used to detect opportunities for possible modifications and chose a plan to carry

out the modification. Each procedure described in Section 5.3 is a plan available to CONSPEC during this phase. In Section 5.3.1 we describe a procedure that merges parallel state machines when a reduction in the number of states can be achieved. In Section 5.3.2 we describe the methods CONSPEC uses for the satisfaction of maximum and minimum time constraints.

The last section of this chapter gives some examples of results provided by the program.

5.1 Data Path Synthesis

Data path synthesis includes module choice, operation scheduling, and module and register allocation and binding. Module choice in the ADAM system is guided by the SLIMOS package for pipelined designs [JPP88] or performed manually. The operation scheduling step of data path synthesis is performed by either MAHA [PPM86] for non-pipelined or Sehwa [PP86] for pipelined designs. Module allocation and binding are performed by MABAL [KP]. The operation schedule determines the number of time slots and the assignment of data flow operations to particular time slots. Multiple minimum and maximum timing constraints may be imposed on the implementation by the specification. These are taken into account by MAHA during data path synthesis, but not by Sehwa or MABAL.

Typically, at least some of the data path modules require control signals. Such modules include certain operators, such as ALU's, as well as multiplexers and registers. Information from the module library can be used to determine the exact values of the control signals required. The operator schedule determines the timing of the signals. This provides all the information needed for control of the data path behavior. No currently existing ADAM program derives this information automatically, so it must be done manually.

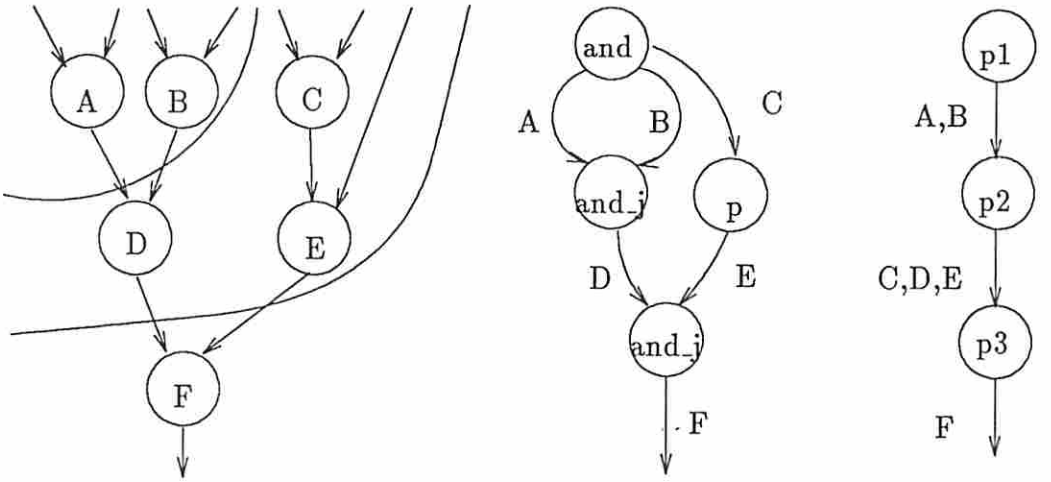
To make this information available to the control specifier, the timing and values of control signals must be incorporated into the timing and control graph derived from the initial specification. The data path clock period must also be represented in the specification. The method followed in incorporating this information into the timing and control graph is described next. This step has not been automated yet, and is currently done manually.

Prior to data path synthesis, the behavior of the unscheduled and unbound data flow operations is represented in the Control and Timing Model as illustrated by an example in Figure 5.1. Each operation (A, B, C, etc.) is bound to its own time interval with unspecified length. The control flow between operations is shown by the type of point connecting the intervals. Operations that execute one after the other because of data dependencies, such as A and D, are bound to a sequence of intervals separated by points. If conditional branches intervene between the operations, an *or*-point precedes the interval to which the conditional operation is bound. In our example, if only one of A or B were to be executed and not both, the and-point would be replaced by an or-point. Operations that are not constrained by data dependencies with each other, such as D and C, are bound to intervals on different parallel branches, represented by and-points and and-joins.

During data path synthesis operations are scheduled into time slots as indicated by the curved lines in Figure 5.1a. MAHA places A and B into the first time step, C, D, and E in the next, and F in the last. (Other schedules could have been possible as well). The initial DDS graph is modified in three ways following data path synthesis, as illustrated by the difference between the graph in Figure 5.1b and c. First, all operations scheduled during the same clock period will be bound to a single interval. This means that sequential intervals are merged into a single interval if all operations bound to them are cascaded together in a single time step. This has happened for operations C and E in our example. In addition, parallel branches reflecting data flow parallelism can be merged into a single path that reflects the operation scheduling.[5.1] Operations A and B are scheduled in the same time slot, so their respective intervals are merged into a single interval, and similarly for operations C, D, and E. Conditional branches are not merged.

The second modification of the Control and Timing Model to reflect data path synthesis is inclusion of control signals. The timing and value of a control signal output is represented by a binding between the structural carrier (wire) on which the control signal is carried, the value to be placed on the wire, and a time interval during which the value should be asserted. Any control signals required by the operator implementing operation A, for example, will be bound to the first interval

[5.1] Parallel branches that represent interfacing behavior as well as data flow behavior are not merged by this step.



A. data flow showing scheduling information

B. initial CT graph

C. final CT graph

Figure 5.1: Effects of Data Path Synthesis on Timing Graph

of the final graph. Third and last, the intervals to which operations are bound are defined as clock periods with a length determined by data path synthesis.

In the case of the partial UNIBUS arbiter example in Figure 3.21 on page 77 of Chapter 3, the single data flow operation less-than-or-equal-to is implemented by gates in the data path. Taking this fragment as a stand-alone design, there are no data path control signals that need to be asserted. The sole effect on the timing graph is the conversion of the first interval, *i1* to a clock interval of length equal to the data path clock.

The clock period for the controller is included in the design specification. The data path clock period must be equal to or an integral multiple of the controller clock period. If a synchronous input is included in the specification, its clock period must also be equal to or an integral multiple of the controller clock.

5.2 State Table Derivation

The modification of the Control and Timing graph to reflect data path decisions completes the representation of behavior for control synthesis. CONSPEC derives a state table specification for a synchronous finite state machine from the control

and timing graph A state table specification consists of a set of states, state transitions for all possible input sequences, and state outputs, which are a function of current state and possibly of inputs as well. Asynchronous inputs are synchronized to the controller clock before being input to the controller. The output of the synchronizing circuits is allowed to change on the same clock edge as synchronous inputs. Their timing will still be indeterminate with respect to state, however. The synchronization process can use standard techniques and is not discussed here.

The state table is derived incrementally during a depth first traversal of the timing graph. The derivation procedure takes as input a *point* of the Timing graph and a *state* name. The behavior associated with the arc or arcs exiting *point* will be implemented by transitions and outputs of *state* (and possibly by a sequence of states starting at *state*). For this reason, *point* is said to mark or correspond to the beginning of *state*. Links between points and corresponding states are used during traversal to determine if a point has been previously visited. Different arcs of the graph pointing to the same point correspond to transitions to the same state. Loops are represented in DDS by pairs of *begin-loop* and *end-loop* points identified by unique subscripts. When CONSPEC encounters a transition to an end-loop point e_i it determines the corresponding begin-loop point b_i, finds the state s linked to b_i, and establishes a transition to s. A sketch of the derivation steps follows; subsequent sections give details on each step.

1. Let *point* = Root point of Control and Timing graph.
2. For each outarc *arc* of *point*, do:
   (a) Determine number of states needed to meet minimum *arc* timing.
   (b) Determine controller outputs associated with *arc*.
   (c) Determine inputs affecting controller behavior during *arc*.
   (d) Specify all state outputs, next state transitions, and their conditions.
   (e) For each point *next* reached in the graph and not yet visited, do:
      i. *point* = *next*
      ii. Go to 2

The points and arcs of the control and timing graph are processed depth-first: at each conditional point one branch is followed until a point is reached that has no out-arcs, or until a previously visited point is reached (because of *or-join* points in the graph different branches may converge on the same point). The system then backs up to the most recently visited conditional point that has unprocessed branches.[5.2] Section 5.2.1 describes how a DDS timing interval classification is used to determine the necessary number of states in Step 2a. Section 5.2.2 describes how CONSPEC determines the proper controller outputs for each state from interval bindings in Step 2b. In Section 5.2.3 we describe how conditional behavior related to synchronous and asynchronous inputs is reflected in state transitions and outputs. In Section 5.2.4, we describe the algorithms used to implement Step 2d.

5.2.1 The Establishment of States

We must determine the proper number of states needed to implement the behavior associated with each arc. There are four basic possibilities: a single arc implemented by zero states, a single arc implemented by a single state, a single arc implemented by multiple states, and multiple sequential arcs implemented by a single state. These cases will be discussed in turn.

5.2.1.1 Single Arc Implemented by Zero States

Certain arcs of the graph, called *glue* arcs, are features of the representation and do not directly correspond to any system behavior. Glue arcs are necessary because not every possible point type has been defined in the DDS, as noted in Appendix A. For example, to end a conditional branch and simultaneously begin a new loop iteration, as in the Arbiter example, an *or_join* point must be followed by an *end-loop* point. These two points are connected by a glue arc. Glue arcs are skipped over in Step 2 of the procedure. Similarly, when assigning state names to a previously unvisited point, the arc following the point is examined. If it is a glue arc, the next point is examined to see if it has been previously visited. If so, the current point is identified with the same state as the point succeeding the glue arc,

[5.2] Parallel branches from *and* points are handled differently, as described in Section 5.3.1.

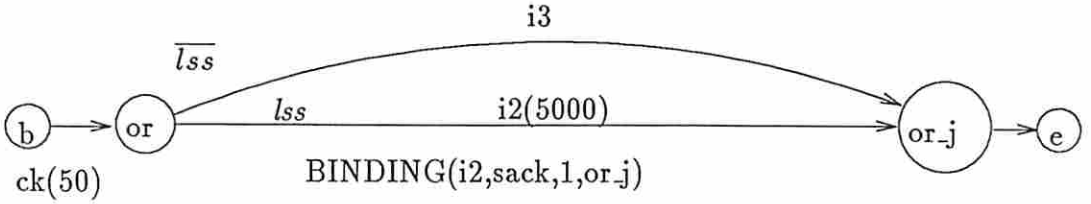


Figure 5.2: Timing Graph of Partial UNIBUS Arbiter Description

and is marked as already visited. If not, both points will be associated with the same new state name.

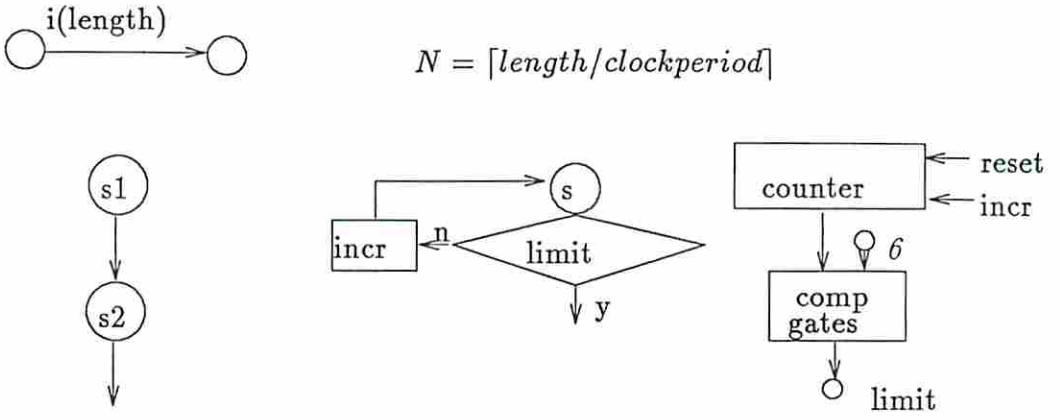
5.2.1.2 Single Arc Implemented by Single State

The second implementation scenario is the designation of a single state to realize the behavior associated with a single interval. This is done in the case of *clock period arcs*, provided that the clock period specified is the same as that used by the controller. If the clock periods are not identical, then the more general implementation method described in the following section is followed. Intervals without a length specified are also assigned a single state. In Figure 5.2, *ck*, which is a clock period of length 50 nsec., and *i3* will each be implemented by a single state.

5.2.1.3 Single Arc Implemented by Multiple States

Interval arcs may be assigned lengths. The number of states, N, necessary to achieve the proper timing is given by $\lceil length/clockperiod \rceil$. If the specified length is less than one clock period, a single state is used to implement the behavior associated with the interval. If the specified length is longer than one clock period, multiple states are specified, each state having the same behavior. If there are any outputs associated with the interval, all states of the sequence have the identical outputs.

Because the behavior of each state is identical, two methods may be used to arrive at the proper number of states: a succession of individual states, or a loop consisting of a single state and a data path counter. The relative desirability of the two methods depends on the implementation style and problem size. Other research on control synthesis at USC [MP] has determined boundary conditions for using a loop versus a sequence of states for PLA implementations. For a



CASE 1: $N \leq 5 \Longrightarrow$ Sequence

CASE 2: $N > 5 \Longrightarrow$ Counter + Loop

Figure 5.3: Single Arc–Multiple States

loop of length one, if the number of iterations N is greater than five, the most area efficient implementation generally is a loop, whereas five iterations or fewer are most efficiently implemented as a sequence of states. This is equivalent to implementing a counter in the controller rather than in the data path.

This heuristic is currently followed by the CONSPEC algorithm, although the system could be expanded to consider heuristics relevant to other implementation styles (see Chapter 4). Figure 5.3 illustrates the concept with an example.

The method used in figures to represent state machines is the Algorithmic State Machine (ASM) chart. Circles are state labels. Conditional transitions and outputs are represented by a diamond inscribed with the conditional input(s). The arcs leaving the diamond are labelled with an input condition. For example, one arc will represent the condition, "input signal is logic 0", and another arc the condition, "input signal is logic 1". State outputs are represented by a box inscribed with the output. If the output is unconditional and is asserted regardless of current inputs, the box immediately follows the state label. If the output is conditional and may change depending on the value of system inputs, then the box is placed after the diamond(s) that specify the input conditions causing assertion of the output.

For Case 1 in Figure 5.3, the required number of states N is less than five, since the clock period is 50 nanoseconds. A sequence of N states, in this case two, is used to achieve the proper interval timing. For Case 2 of Figure 5.3, N is seven, again assuming a clock period of 50 nanoseconds. Because N is greater than five, a loop is

specified in the state machine and a counter is added to the data path. The single state loop must be exited when the value contained in the counter is equal to *N-1* (the count begins at zero), in this case six. Therefore, CONSPEC adds extra data path hardware in the form of gates to compare the counter value to the constant value *N-1*. The output of these gates is input to the controller and conditions the state transition out of the loop. Two control signals are necessary to achieve the proper data path behavior. A control signal to clear the register is asserted during all states immediately preceding the loop (not shown), and an increment signal is asserted during the loop state. In certain cases, an interval may have infinite length specified. In this case a single state loop is used, but no counter is added to the data path, since the number of loop iterations is unbounded. Exit may be defined by the conditional action of asynchronous signals for such specifications.

Referring to the arbiter example in Figure 5.2, interval *i2* requires 5000/50 = 100 state periods for its implementation. This will be achieved using a state loop. In Section 5.2.3 we will show, however, why the conditional branch preceding *i2* results in a modification to the implementation. In Section 5.2.3.2 we show how a loop implementing an infinite interval may have an exit condition if the specification includes an interrupt signal on the interval.

5.2.1.4 Multiple Intervals Implemented by Single State

A third implementation possibility is the placing of behavior bound to multiple sequential arcs into a single state. Data path synthesis can have this effect, since it may assign a clock period large enough to cascade several operations that are bound to sequential intervals in the original specification. Operations C and E of Figure 5.1 are bound to a single time interval for this reason. The CONSPEC algorithm does not deal with this, however, since the effects of data path synthesis are already incorporated into the Control and Timing graph.

Sequencing requirements cannot be verified by examining data dependencies alone, since external constraints may be responsible, therefore CONSPEC assumes that the sequencing requirements specified by the graph are absolute and cannot be violated. For this reason, there can be no arbitrary decision to place two sequential operations into a single time step, even if the combined specified lengths are less than a single clock period. Section 5.3.2 describes one method of accomplishing

this that does not violate sequencing requirements and can be used only in special cases.

There is only one standard configuration in which sequential intervals and their associated behavior are implemented in a single state by CONSPEC, and that is when *clock phase* arcs, representing the low and high phases of the clock period, are used in a specification. Some behavioral descriptions utilize clock phase arcs to specify on which clock phase input signals are allowed to change and when outputs signals should be asserted. The Frame Counter example given in the previous chapter is an example of such a specification. For such cases, CONSPEC creates a single state for each pair of clock phase arcs.

5.2.2 Controller Outputs

The output of values is specified in the DDS by a binding between a value defined in the data flow graph, the time interval during which it is to be output, and the structural carrier on which the value should be asserted. There are three types of output bindings, the latter two of which are implemented as controller outputs.

The first specifies the output of data flow values that were assigned during data path synthesis to registers. This is the case, for example, for value a in the statement, "value a (stored in register x), is to be placed on the bus at time t." Value a is not an output of the controller, so this binding is not relevant.

A second type of output binding refers to the control signals of data path elements such as register x in the example above. The value and timing of signals in this category are determined by the module allocation and binding step of data path synthesis. Data path modules of various kinds require control signals, including multiplexers, registers, interface lines (e.g., tristate), and some operators such as ALU's. These signals must be produced by the controller and so their associated bindings are used in determination of state outputs.

The third type of output binding refers to binary constants, such as 0 or 101_2. Binary constants are often a part of specifications including interfacing behavior, and function in a handshaking or control capacity during communication with external devices. If the number of bits are greater than some limit the value will be stored in a data path register and output through a multiplexer. Otherwise, for

binary constants with fewer bits they are produced directly by the controller as a state output. In the current implementation it is assumed that the limit is never exceeded, so the controller always produces the outputs. For future extensions this limit should be user specified with some default value such as four.

For each interval being processed, therefore, the outputs that must be produced by the controller are determined from output bindings on the interval. In the next section we will explain the conditions under which outputs must be produced.

5.2.3 Controller Inputs

Controller inputs determine the course of conditional system behavior. In the Control and Timing Model of the DDS there are two types of conditional behavior that may be represented: branches based on the values of synchronous signals and branches governed by asynchronous signal events. The term synchronous is used to indicate a signal whose timing is precisely defined in relation to the system clock. Synchronous signals arise from either the data path or from an external system with a known clock. Asynchronous signals have indeterminate timing in relation to other system events. They are produced in the external environment: examples include reset signals and other signal interactions between independent processors. Asynchronous signals must be synchronized to the system clock before being input to the controller. Synchronization is performed to avoid signal races and possible errors in the state machine. The timing of these synchronized signals is still indeterminate with respect to state, however. For this reason, and because their representation is different from that of synchronous signals, we continue to refer to them as asynchronous signals.

The next two sections describe the methods used to implement the behavior described by synchronous predicate and asynchronous predicate bindings. The following section then gives an outline of the algorithm used by CONSPEC to determine the conditions for state outputs and next state transitions, given an arbitrary combination of arc types, lengths, and predicate bindings.

5.2.3.1 Synchronous Input Signals

The conditional effects of synchronous input signals are represented in the DDS Control and Timing graph by an or-point, as described in Section 3.3.4.2 of the previous chapter. This point has a single incoming arc and multiple outgoing arcs corresponding to the various branches. The input signals affecting the branch are represented by a *synchronous predicate binding* attached to each out-going branch. The predicate is a Boolean expression that may be composed of multiple input values. Only one of the predicates attached to the outgoing arcs can be true at a time during processing.

Conditional branches based on synchronous events or signals are either/or situations. If conditions are right, a particular branch is taken, otherwise it is not taken. Once this choice is made, the conditions do not need to be rechecked. Correspondingly, in the DDS the semantics of the or-point stipulate that when the point is reached in processing, a branch is made based on the value of the synchronous predicate. The arc whose synchronous predicate is true is the one followed. Synchronous predicate bindings therefore establish the values of any synchronous input signals that must be true *at the beginning of the interval* for the events associated with the interval to be executed. The input signals may change once the branch choice is made, possibly before branch processing is complete.

The transitions and outputs of a state associated with the arcs following an or-point must therefore be conditional on the synchronous input values. To accomplish this, CONSPEC establishes a value reflecting the conditional effects of synchronous input signals for each arc of the graph. This value, called *cond1*, is equal to the synchronous predicate bound to the interval. Synchronous predicates are in the form of a list of input values. For a value of name *val*, if a positive logic level must exist for that branch being taken, the corresponding element is *val*. Conversely, if a negative logic level leads to the branch being chosen, the corresponding element in the synchronous predicate is *n(val)*. Several such values may be included in a single predicate. If there is no synchronous predicate binding, cond1 is nil.

The effects of synchronous inputs on state transitions and outputs is discussed for three cases in turn: for intervals implemented by a single state, by multiple states, and for two clock phase intervals implemented by a single state.

5.2.3.1.1 One Interval Implemented By a Single State This is the simplest case, illustrated in Figure 5.4. Each timing interval contained in the two graphs can be implemented within a single state period. We first describe the derivation for the graph on the left. Upon reaching the or-point, which is linked to the name State A, CONSPEC first examines the arc on the left branch and determines that a single state should be used for the implementation. Output bindings are examined, followed by input conditions. *Cond1* for this interval is equal to [op], since that is the synchronous predicate bound to the interval. A name is established for the next state, State B, and linked to the end point of the interval. A transition from State A to State B under condition cond1 is established. Outputs bound to the interval are also asserted in State A under condition cond1. State B is then processed as the derivation moves on.

When processing of the left branch of the graph terminates, CONSPEC returns to the arc on the right branch and determines that a single state should be used for the implementation. *Cond1* for this interval is equal to [n(op)], since that is the synchronous predicate bound to the interval. A name is established for the next state, State C, and linked to the end point of the interval. A transition from State A to State C under condition cond1 is established; State A asserts any outputs bound to the interval under cond1 as well. This step is followed by the processing of State C. For the graph at the left, therefore, State A is assigned to implement the two intervals exiting the or-point. State A has two conditional transitions: under synchronous input condition $op = 1$, the transition is to State B. Under condition $op = 0$ the transition is to State C.

We do not assume a lifetime for the input value *op* that extends beyond the interval in which it is tested. The separate states B and C serve therefore to "remember" what the input value was. This is a necessity when branching characteristics show that subsequent behavior depends on such knowledge.

For the graph on the right of Figure 5.4, upon reaching the or-point, linked to the name State A, CONSPEC first examines the arc on the left branch. A transition is specified from State A to State B under cond1 = [n(less)], and State B is processed. When processing of that branch terminates, CONSPEC returns to the arc on the right branch. This time, when determining a name for the point *or_j*, it is found to be already linked to State B. A transition is therefore specified from

5.2.3.1.1 One Interval Implemented By a Single State This is the simplest case, illustrated in Figure 5.4. Each timing interval contained in the two graphs can be implemented within a single state period. We first describe the derivation for the graph on the left. Upon reaching the or-point, which is linked to the name State A, CONSPEC first examines the arc on the left branch and determines that a single state should be used for the implementation. Output bindings are examined, followed by input conditions. *Cond1* for this interval is equal to [op], since that is the synchronous predicate bound to the interval. A name is established for the next state, State B, and linked to the end point of the interval. A transition from State A to State B under condition cond1 is established. Outputs bound to the interval are also asserted in State A under condition cond1. State B is then processed as the derivation moves on.

When processing of the left branch of the graph terminates, CONSPEC returns to the arc on the right branch and determines that a single state should be used for the implementation. *Cond1* for this interval is equal to [n(op)], since that is the synchronous predicate bound to the interval. A name is established for the next state, State C, and linked to the end point of the interval. A transition from State A to State C under condition cond1 is established; State A asserts any outputs bound to the interval under cond1 as well. This step is followed by the processing of State C. For the graph at the left, therefore, State A is assigned to implement the two intervals exiting the or-point. State A has two conditional transitions: under synchronous input condition $op = 1$, the transition is to State B. Under condition $op = 0$ the transition is to State C.

We do not assume a lifetime for the input value *op* that extends beyond the interval in which it is tested. The separate states B and C serve therefore to "remember" what the input value was. This is a necessity when branching characteristics show that subsequent behavior depends on such knowledge.

For the graph on the right of Figure 5.4, upon reaching the or-point, linked to the name State A, CONSPEC first examines the arc on the left branch. A transition is specified from State A to State B under cond1 = [n(less)], and State B is processed. When processing of that branch terminates, CONSPEC returns to the arc on the right branch. This time, when determining a name for the point *or_j*, it is found to be already linked to State B. A transition is therefore specified from

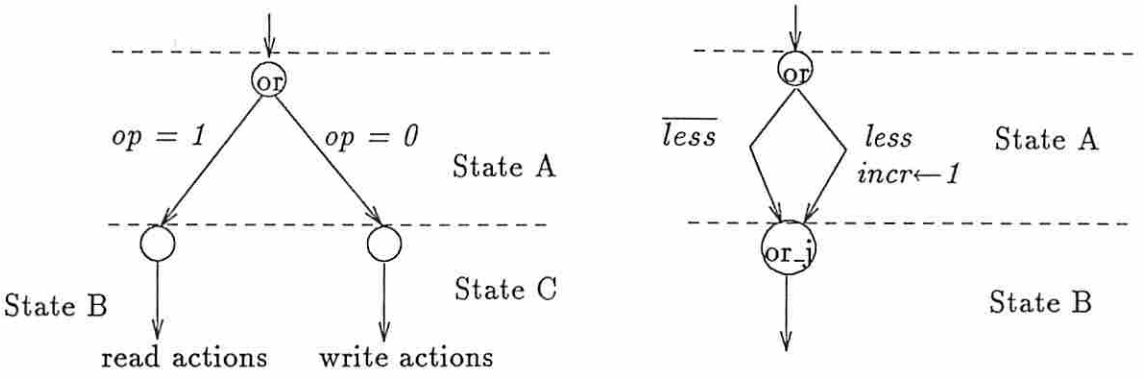


Figure 5.4: DDS Conditional Branches and State Transitions

State A to State B under cond1 = [less], and the output *incr* is asserted by State A under cond1 as well. Processing on this branch then terminates because point *or_j* was previously visited. In this case State A is also assigned to implement two intervals exiting an or-point. The next state transition from State A is to State B regardless of input value, and is therefore unconditional. However, output *incr* is conditionally asserted.

5.2.3.1.2 One Interval Implemented by Multiple States If it takes more than a single state to implement an interval succeeding an or-point (because a length exceeding the clock period is specified), the value of the synchronous input should be checked only during the first state period. This is because synchronous inputs may change at some time during the interval and cause an incorrect transition. For our synchronous implementation, we assume that the input conditions remain valid long enough to be checked during a single state period. Synchronous inputs arise from either the data path or from an external source with a known clock period. The initial CONSPEC clock period is taken to be the smallest clock period of all synchronous system behavior in the specification, including data path and synchronous input signals. The initial controller clock period may be modified only by making it smaller, as will be seen in Section 5.3.2. For this reason, a synchronous input arising from either the data path or the external environment will be valid for at least one controller clock period, and therefore our assumption is correct.

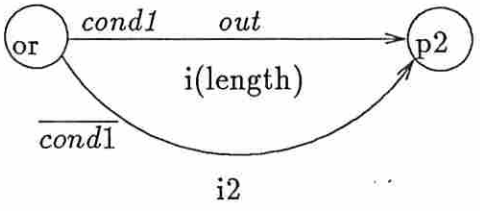
Recall from Section 5.2.1 that an interval with length specified is implemented by $\lceil length/clockperiod \rceil = N$ states. This is achieved by using either a single state

loop and data path counter, or by a sequence of N states, depending on the value of N. When the interval is a branch of an or-point, the first state of the sequence must have transitions conditional on the synchronous inputs. The sequence is entered only under input condition cond1. Subsequent state transitions in the sequence are not conditional on cond1, because the synchronous inputs are not checked again by any other state in the sequence. Figure 5.5b shows the implementation of an interval using a sequence of three states. The transitions and outputs of *s1* are conditional on the value *cond1*. Transitions of states *s2* and *s3*, however, are unconditional. The value *out* bound to interval *i* is only asserted during *s1* if *cond1* is true, but is asserted unconditionally in subsequent states of the sequence. Under input condition $\overline{cond1}$, *s1* implements any events bound to interval *i2*.

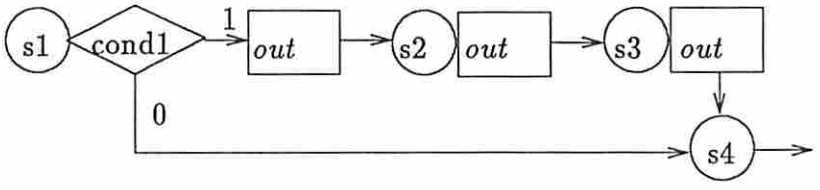
Correspondingly, if a state loop is to be used to implement an interval following an or-point, the loop should be entered only if the synchronous input conditions are right. This requires the introduction of a second state to the implementation strategy as shown in Figure 5.5c. During *s1* transitions are made based on the value of *cond1*. If true, a loop is entered. The transition of *s1* under condition *cond1* implements one state period of the required time interval. The loop is therefore iterated one time less than the total number of state periods required to implement *length*. The transition made by *s1* under condition $\overline{cond1}$ implements interval *i2*.

5.2.3.1.3 Clock Phase Arcs Implemented by Single State We assume that all input signals change value on the same phase or edge of the clock, and that state transitions are scheduled to occur on the opposite edge from input transitions, in the middle of the input value's lifetime. Conditional state outputs that depend on the value of an input as well as state are therefore (potentially) asserted in the middle of the state, when the input value is allowed to change.

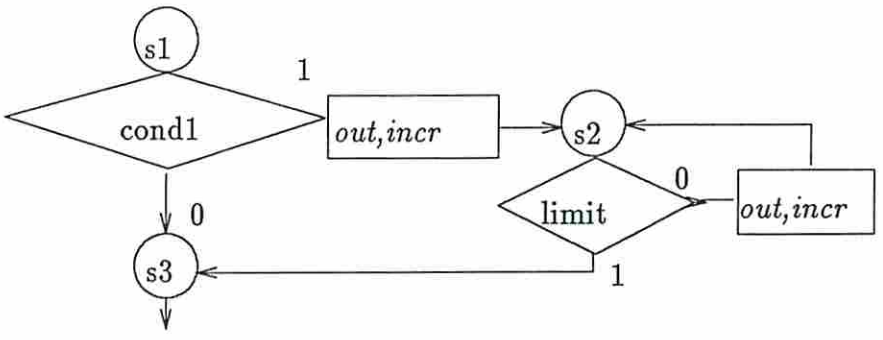
Under these assumptions, which are commonly made to guarantee safe performance of state machines, in a DDS timing graph utilizing phase arcs there will be at most one or-point in each graph segment consisting of one high and one low clock phase arc. If the input signals are allowed to change on the rising edge of the clock, the or-point immediately precedes the high phase arc. This signifies that the value of the input is tested during the high phase of the clock, and a branch



a. Specification



b. State Sequence



c. State Loop plus Data Path Counter

Figure 5.5: Effects of Synchronous Inputs on State Diagram

taken accordingly. Conversely, if the input signals are allowed to change on the falling edge of the clock, the or-point immediately precedes the low phase arc.

The derivation of a state table from such a specification is virtually identical to that described under Paragraph 5.2.3.1.1. The Frame Counter in Figure 5.6 illustrates the process. The derivation procedure is called initially with point b_j and state *s1*. The arc following b_j is a phase arc, so CONSPEC moves on to the next point, $or1_j$, staying in *s1*. The conditional branches following this point represent conditional transitions of *s1* that depend on the synchronous inputs *C* and *D*. The control synthesis for this example is described in more detail in Section 5.4.2.

In some cases, the interval following the or-point may be an interval arc with length specified instead of a phase arc. This would be true for the specification: "If x (which changes on the rising edge of the clock) has logic value 1, delay 5000 ns." In this case, multiple states would be used for the implementation in a manner analogous to that described under Paragraph 5.2.3.1.2.

5.2.3.2 Asynchronous Input Signals

The exact timing of asynchronous input signals is unknown. The only way to detect the occurrence of an asynchronous event therefore is by a change in value, or by the detection of a specific value on defined lines. A synchronous system must "catch" an asynchronous signal by testing its value once per clock period. If the signal occurs after the clock edge used by the synchronizing circuits,[5.3] it will go undetected until the next state period. If it goes away before the next synchronizing clock period edge, it will be lost completely. The synchronizing circuit must therefore have an adequately fast clock period. The specification indicates the timing necessary to avoid missing signals. If speed demands are very great, a synchronous specification may be unable to perform adequately. Such specifications require a different design approach, such as the so-called *largely synchronous* state machine approach, in which some controller inputs are asynchronous though the overall execution is synchronous.

In the DDS, conditional branching dependent upon asynchronous inputs is represented by an *asynchronous predicate binding* represented as Binding(interval,

[5.3]Recall that asynchronous inputs are passed through synchronizing circuits before being input to the controller.

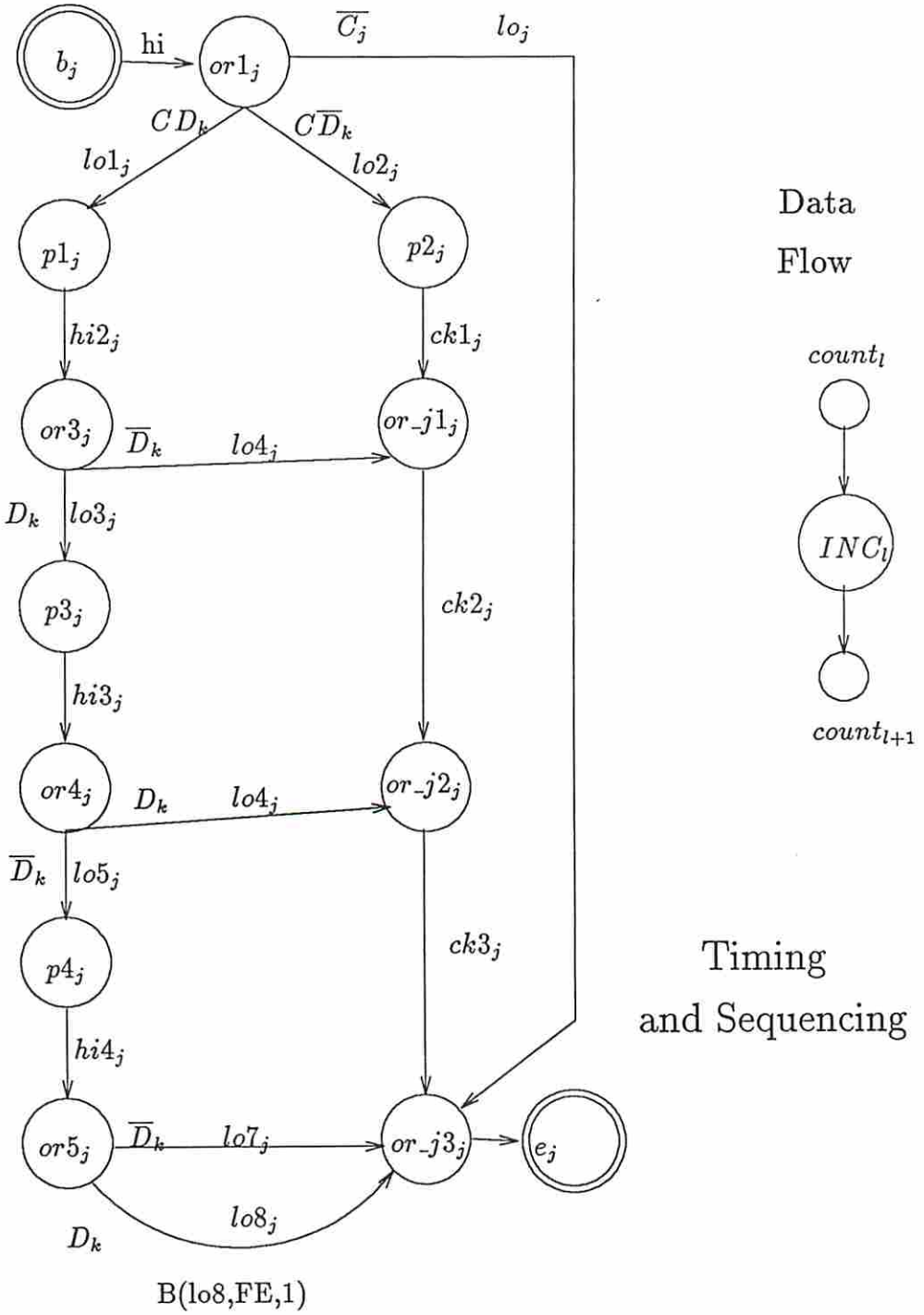


Figure 5.6: DDS Timing Subspace Including Control Signals

carrier, value, point).[5.4]. The predicate specifies a timing graph *point* to which a branch should be made if the data flow *value* is detected on the physical *carrier* at any time during the timing *interval.* If a predicate becomes true at any time during the interval the specified branch is made. Otherwise the branch is not made and processing continues at the next interval, if any. Any number of asynchronous predicates can be bound to a single interval, but they must be mutually exclusive. Any outputs associated with *interval* are asserted only so long as no asynchronous predicate is asserted. This is because the assertion of an asynchronous predicate defines a termination of the behavior associated with *interval*, and the beginning of some other behavior.

The transitions and outputs of a state associated with an arc bound to an asynchronous predicate must be conditional on the asynchronous input values. To accomplish this CONSPEC establishes two values reflecting the conditional effects of asynchronous input signals for each arc of the graph. The first is called *cond2*; it defines the values of all asynchronous inputs that must exist so that *no* branch occurs and events bound to *arc* are executed. It is a list, defined in the following way:

> For each asynchronous predicate *Binding(arc,signal,value,point)* one element of Cond2 is defined as *n(signal)* if value=1 and as *signal* if value=0. If there are no asynchronous predicates, Cond2 is nil.

The second value reflecting the conditional effects of asynchronous inputs is actually a group of values, one for each asynchronous predicate bound to an arc. Each value, called *cond3* defines a single asynchronous input condition under which a *branch* occurs and events bound to *arc* are *not* executed. Each cond3 is also a list, with a single element whose value is calculated in the opposite way as that given for cond2:

> For asynchronous predicate binding number *i*: *Binding(arc,signal,value,point)*, Cond3(i), is defined as *n(signal)* if value=0 and as *signal* if value=1.
> If there are no asynchronous predicates, there is no cond3.

[5.4]In figures an alternate representation is sometimes used: carrier=value→branch-point, indicating that if *value* is detected on *carrier*, a branch should be made to *branch-point*

If, for example, there are two predicate bindings referring to asynchronous inputs P1 and P2 that are asserted high, cond2 is [n(P1),n(P2)], indicating that no branch is made under input conditions (P1=0 and P2=0). Cond3(1) is [P1], corresponding to input condition (P1=1 and P2=0), and cond3(2) is [P2], corresponding to input condition (P1=0 and P2=1). Each cond3 value indicates an input condition under which outputs associated with the interval are no longer asserted, and a branch is made to *point* of the corresponding predicate binding. The condition (P1=1 and P2=1) is not allowed to occur; it is treated the same as the non-asserted condition.

CONSPEC always assigns a new state to implement the behavior of the arc following *point* to which a branch is made. This is a conservative implementation, since in some special cases it may be possible to implement behavior associated with that arc by asserting conditional outputs in the same state during which the asynchronous input is detected.

Each asynchronous predicate bound to an interval represents a branch through the timing graph that must be traversed by CONSPEC. Step 2e of the derivation steps given in Section 5.2 specifies a recursive call of the derivation for each point reached in the graph. One of these points will be the end point of the current interval. Other points will be branch points specified by asynchronous predicates bound to the interval, one per binding, if any. The recursion ends when the derivation procedure is called with a point that has no out-arcs, or when all of the points reached have been previously visited.

5.2.3.2.1 One Interval Implemented by a Single State If an interval with asynchronous predicate bindings is implemented by a single state, *s1*, and the endpoint of the interval corresponds to *s2*, then *s1* transitions to *s2* under condition Cond2. Outputs bound to the interval are asserted by *s1* under Cond2 also. For asynchronous predicate binding number *i*, *s1* transitions to the corresponding state of branch-point(i) under condition Cond3(i); for these transitions *s1* has no outputs. The same description applies to the transitions and outputs of a state that implements the events bound to the two phase arcs representing a clock period.

5.2.3.2.2 One Interval Implemented by Multiple States If the interval is implemented by multiple states a branch is made if the predicate is asserted during any state.[5.5] The transition from one state to the next in the sequence is conditional on Cond2 being true. Every state of the sequence also has one transition for each asynchronous predicate binding number *i*, under condition Cond3(i), to the corresponding branch state.

An example of this is shown in Figure 5.7. The interval *i* is bound to an asynchronous predicate, *reset*. If reset is asserted at any time during *i*, a branch is to be made to point *or_j*, which defines subsequent processing. During the duration of the interval, the value 1 is to be placed on carrier *out* as specified by a binding. The two implementation possibilities, a single state loop and a state sequence, are shown.[5.6] The transition conditions for each state are identical, so it is not necessary to introduce a second state to the single state loop as done for synchronous predicate bindings. *Lim* is a value that reflects a test on the contents of a data path counter. When *lim* becomes true, the required number of states has been reached and the loop is exited. Note that *out* is asserted during *s1* only under condition reset=0, that is, only if the predicate is *not* true.

It was noted in Section 5.2.1.3 that intervals with infinite length, which are implemented by a single state loop, can have an exit condition defined by an asynchronous input. This corresponds to an indefinite *wait* for the assertion of a asynchronous signal, with no time-out condition. An example is shown in Figure 5.8. State *s1* begins at point *p1* and state *s2* at point *or_j*. The asynchronous predicate binding along with the infinite time length indicates that the system should remain within interval *i* until the asynchronous predicate *signal* is asserted. The state machine thus loops in *s1* until *signal* is asserted, and then moves to state *s2*.

5.2.4 Establishment of State Outputs and Transitions

The derivation of a state table from a timing graph is performed one arc of the graph at a time, during a depth-first traversal of the graph. The method used

[5.5] Note that this is unlike the treatment of synchronous predicate bindings, in which only the first state has transitions conditional on the predicate.

[5.6] One of these would be chosen based on the number of states required.

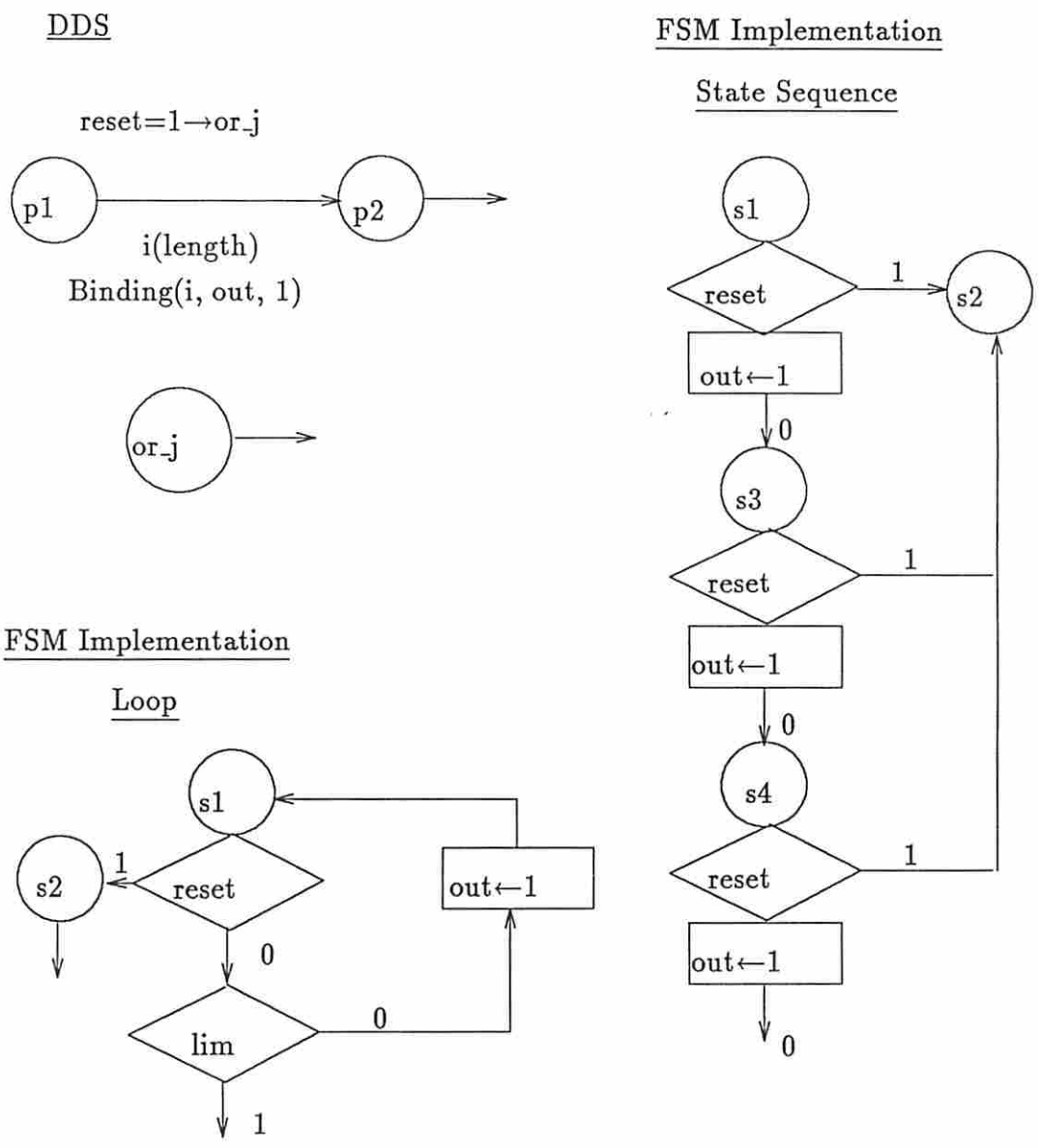


Figure 5.7: Effects of Asynchronous Inputs on Transitions and Outputs

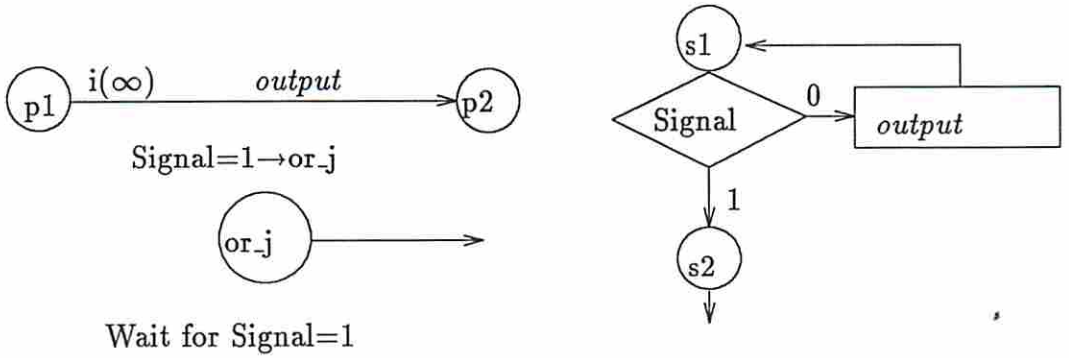


Figure 5.8: Asynchronous Exit Condition On Infinite Loop

to determine a state implementation for each arc was outlined at the beginning of Section 5.2. In Sections 5.2.1 through 5.2.3 we discussed in some detail the methods used by that procedure at each step of the derivation. Here we give a description of Step 2d, which is to specify all state outputs, next state transitions under the proper conditions.

Processing is kept simple by the method of defining input conditions, reiterated here:

1. cond1 = synchronous predicate binding
2. cond2 = values of asynchronous predicates *not* resulting in branch
3. cond3(i) = value of asynchronous-predicate(i) resulting in branch
4. cond4 = cond1 $\cup$ cond2

Conditions 1,2, and 4 are determined for each interval, regardless of whether or not predicate bindings exist. If no synchronous inputs affect system processing during a particular interval, cond1 is nil. If no asynchronous inputs affect processing, cond2 is nil and there are no cond3's defined. Conversely, an interval may have both synchronous and asynchronous predicate bindings, that is, system behavior may depend on the values of both synchronous and asynchronous inputs during the same interval. Cond4 reflects the combined effects of synchronous and asynchronous inputs. If the behavior associated with an arc is unconditional and unaffected by any input signals, cond4 is nil.

For some arbitrary arc *a1*, bounded by point *p1* corresponding to state *s1* and point *p2* corresponding to state *sn*, the following rules are followed by CONSPEC in specifying state outputs and next state transitions, regardless of the particular predicates actually bound to the arc.

1. For the first state assigned to implement the behavior of *a1*:
   (a) For each controller output *out* associated with *a1*:
   Assert *out* under input condition *cond4*.
   (b) Transition to the next state under input conditions *cond4*
   (c) For each asynchronous predicate *Binding(a1,signal, value, point)*:
   Transition to branch state under conditions *Cond1* $\wedge$ *Cond3*

2. For any subsequent states created to implement *a1*:

   (a) For each controller output *out* associated with *a1*:
   Assert *out* under input condition *cond2*

   (b) Transition to next state under input conditions *cond2*

   (c) For each asynchronous predicate *Binding(a1,signal, value, point)*: Transition to branch state under condition *signal=value*

3. If a state loop is used to implement the interval:

   (a) If cond2 $\neq$ cond4, introduce extra state *s-loop*, else let *s-loop* = *s1*.

   (b) s-loop transitions to s-loop and asserts outputs associated with interval under condition $cond2 \cdot \overline{limit}$

   (c) *s-loop* transitions to *sn* under input condition *limit*.

A state machine may have several inputs, not all of which are tested during each state transition. The conditions cond1 through cond4 do not explicitly mention all control inputs, but only those tested during the current state. When printing out the state table at the end of the derivation, if transitions and outputs for a state do not not depend on a particular input's value, *don't care* is given as the value of that input for all transitions of the state rather than explicitly specifying all possible combinations.[5.7] Unconditional outputs and transitions are specified by null conditions (empty set): all input values are specified as don't cares in that case. In Section 5.4.1 we show the control synthesis method for the UNIBUS arbiter example, illustrating the effect of a combination of asynchronous and synchronous predicate bindings on a single arc.

5.3 State Table Modification

In Chapter 4 we discussed a general, goal based design approach to managing the application of algorithms that modify the state table. The correct application of such modifications can potentially improve the design and perform optimization. In this chapter we describe three modification algorithms that have been

[5.7] Don't cares are specified as x in the state table.

implemented. The first is designed to merge separate state machines arising from parallel branches. The second is a method for the satisfaction of minimum time constraints, and the third is an algorithm that attempts to satisfy violated maximum time constraints.

5.3.1 Parallel Branches

Parallel branches are represented in the DDS by pairs of and-points and and-join points.[5.8] CONSPEC implements parallel branches using one of two methods. First, each branch can be implemented by a separate finite state machine with an identical clock period. Second, under certain conditions the separate state machines can be merged and incorporated into the main state machine. In the following section we describe the separate machine technique. In Section 5.10 we describe under which conditions the machines may be merged and the method used to do so.

5.3.1.1 Separate State Machines

A two signal handshake is used between the main state machine and each branch machine. The main machine asserts a single initiation signal that is input to all branch machines. Each branch machine asserts a unique termination signal during its last processing state; the termination signal is then asserted in a final state loop until the initiation signal is deasserted. The main machine deasserts the initiation signal when the termination signals of all branch machines are positive; it then proceeds with the actions bound to the arc following the *and_join* point. The handshaking is performed in such a way that no execution delays occur because of the exchange of signals.

We refer to an example shown in Figure 5.9 to clarify the handshaking protocol. The partial DDS representation of a parallel branch is shown at the top of the figure. There could be any number of branches; two are shown in this example. At the bottom of the figure is the state machine implementation using a single machine for each branch, along with the main machine that executes the actions

[5.8] Recall that the and_join point is similar to a co-end, in which all branches must terminate before proceeding.

preceding and those following the branch. The relation between arcs of the DDS graph and state transitions is indicated by annotating state transitions with the corresponding DDS arc in italics. The first arc of branch1 is *i2* and the final arc is *im*. The first arc of branch2 is *i3* and the final arc is *in*.

The transition from *s1* to *s2* in the main machine implements the behavior of arc *i1*. The initiation signal *init* is asserted and a loop is executed within *s2* in the main machine until both termination signals *term0* and *term1* have been asserted by the branch machines. The first state of the machine implementing branch1, *s3*, loops under condition $init = 0$. Under condition $init = 1$, other transitions of *s3*, of which there could be several, implement the actions of the first arc of the branch, *i2*. During the first state that the main machine asserts *init*, therefore, the actions associated with interval *i2* are executed. Similarly, for the machine implementing branch2, *sk* loops under condition $init = 0$. Under condition $init = 1$, other transitions of *sk* implement the actions of the first arc of the branch, *i3*.

Termination signals are asserted by each branch machine when executing the events of the last DDS arc of the branch. For example, the last arc of branch1 is *im* (there could be multiple such arcs due to or-joins). During the corresponding state transition of the branch1 machine, *term0* is asserted. The machine then enters *sj* where it continues to assert *term0* and loops, waiting for the deassertion of *init*. If all branch machines were to assert termination signals during the same state, in the state period immediately following, the main machine would proceed with the actions associated with arc *ip*. If the branch machines terminate at different times, the main machine resumes processing in the state period following the last machine's termination. In this way no state periods are wasted in exchanging the hand shake signals. The branch machines do require one state period after termination, however, before the branch can be reentered.

The processing of parallel branches deviates from the steps described at the beginning of Section 5.2. This is because the multiple out-arcs of the and-point are implemented by one state for each arc rather than by a single state for all, as would be the case for an or-point. When CONSPEC encounters an and-point, the state implementation of each of its outgoing arcs is decoupled from the point. The subsequent state table derivation along each branch follows the normal procedure until the corresponding and_join point is reached. At that point the handshaking

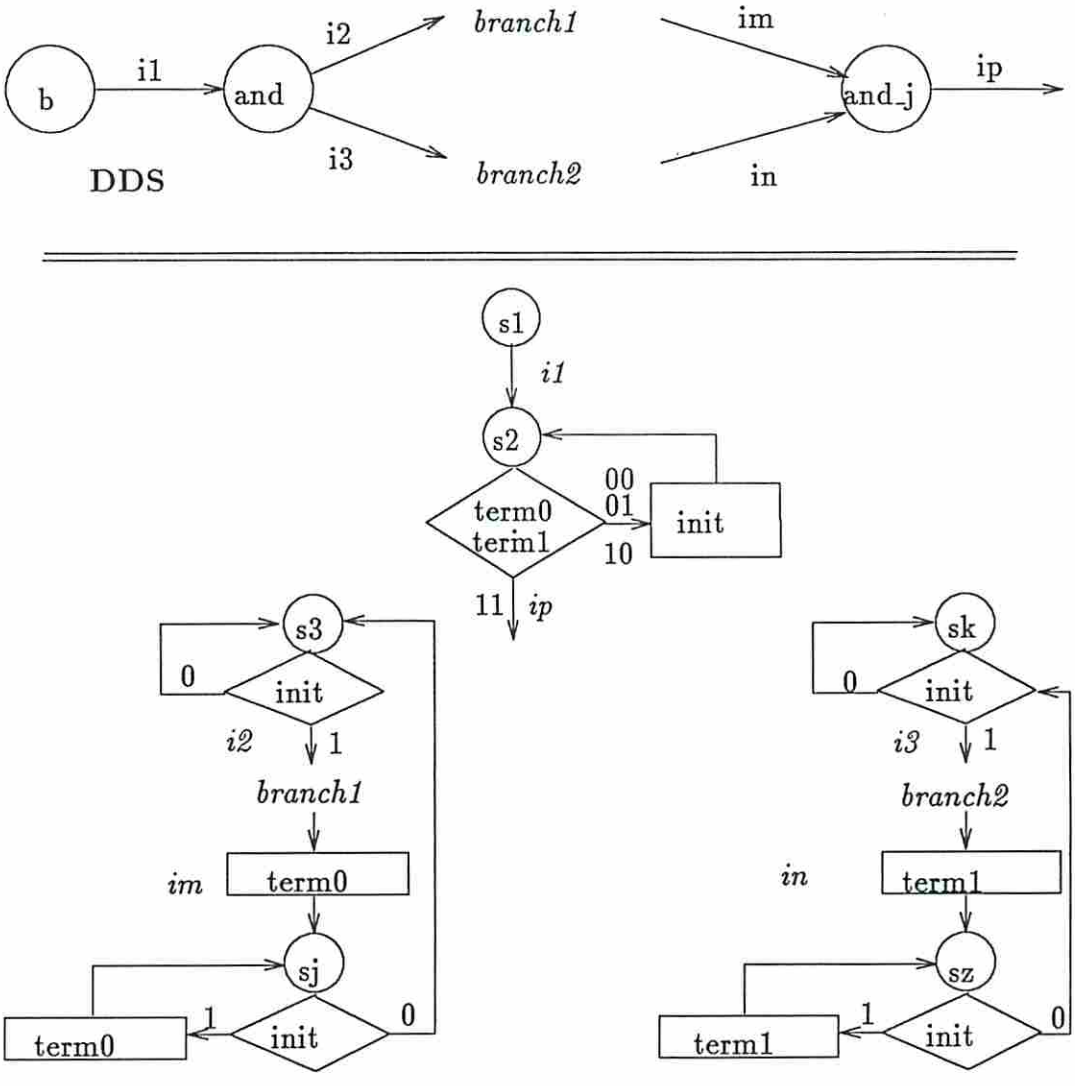


Figure 5.9: Separate Machine Implementation of Parallel Branches

protocol is set up for the main and branch machines, and the derivation proceeds as usual for the rest of the graph.

5.3.1.2 Single State Machine

CONSPEC initially creates a separate state machine for each parallel branch, as described in the preceding section. It is desirable in some cases, however, to implement parallel branches using a single state machine. Because the merging of state machines involves finding the cartesian product of all input conditions affecting branch machine transitions, a combinatoric explosion in the number of states may occur. Nevertheless, if there are many unconditional transitions and a low branching factor, a reduction in the number of states and transitions may be achieved. An additional advantage is the elimination of two handshaking signals lines from each machine. A single machine implementation is preferable to a multiple machine implementation when the following objectives can be met:

1. The number of states in the single machine does not exceed the combined number of states using separate machines.

2. The size of the single machine does not exceed some established limit related to implementation considerations for single state machines.

Before attempting to merge multiple state machines arising from a single parallel branch, CONSPEC checks the state tables. If any branch machines are so large that they equal or exceed a limit on state machine size set by the user, a merge will not be attempted. In addition, if there are internal loops on any branch machine, a merge is not performed. This is because such loops will result in very complex state transitions in a single machine implementation, particularly since loops on separate branches could overlap in arbitrary ways, and because the number of iterations may not be determinate. Otherwise, CONSPEC proceeds to derive the state table created by the merging of all branch machines into a single machine. The resulting number of states is compared to the combined number of states for the multiple machine implementation. The single machine implementation is chosen if there is no increase in the number of states and if it does not exceed the limit on the total number of states for a single machine.

The single state machine created by the merging process can either be combined with the main state machine or established as a separate machine initiated by the main machine. The latter choice is made if the combined size exceeds the limit on the number of states that a single machine may have.

CONSPEC creates a single state machine implementation from multiple parallel branches by merging the transitions and outputs of one state from each branch machine at each *level.* Level is a measure of proximity to the start state in terms of the number of state transitions. At the first level the start states of all machine are combined, at the second level we combine one state from each machine that is reachable by a single transition from its start state, at the third level we look at states reachable by two transitions, etc. States are chosen for merging during a simultaneous depth-first traversal of all branch machines. By the end of the parallel traversal, the paths through the branch machines have been traversed and states have been merged in all possible combinations. The number of transitions for each merged state is the product of the transitions of its constituent branch states, with conditions given by the cartesian product of the branch state conditions. A link is made between the merged state name and the branch machine states constituting it; if a combination of states already has such a link, then it has been processed earlier and the current traversal is terminated. This is necessary because there can be more than one transition to a single state.

A sketch of the merging process follows:

1. Initialize *branch-states* = $[start_1, start_2, \ldots, start_i]$, where $start_j$ is the start state of the jth branch machine, and i is the total number of branch machines.

2. Initialize *Merged* = new state name. Link *Merged* to *branch-states.*

3. For each element $state_j$ of branch-states let $trans_j$ be the set of all possible next state transitions in the form [cond, nextstate].[5.9]

4. Let *combinations* be the set product between all $trans_j$ from $j = 1, i$.[5.10]

5. For each element of *combinations*, do:

[5.9] Inputs to the controller that have no conditional effect during $state_j$ are not included in *cond.*

[5.10] Each element of *combinations* is a set containing one element from each $trans_j$.

(a) Let *nextstates* = set of all $nextstate_j$ for $j = 1, i$

(b) Let *conds* = set of all corresponding $cond_j$ for $j = 1, i$[5.11]

(c) For each element $state_j$ of branch-states, any outputs asserted under condition $cond_j$ are asserted by *Merged* under condition *conds.*

(d) If *nextstates* is already linked to a state name *State*,
Then *NextMerge* = *State*
Else *NextMerge* = new state name; Link *NextMerge* to *nextstates.*

(e) Assert a transition from *Merged* to *NextMerge* under condition *conds.*

(f) If *NextMerge* has not been previously visited, do:

i. Merged ← NextMerge

ii. branch-states ← nextstates

iii. Go to 3

The behavior of one state from each branch machine is combined and implemented by a single state, *Merged.* Each element of *combinations* contains one transition from each branch state, which together define one transition of *Merged.* A link is made between *Merged* and the list of branch machine states; once a particular combination of branch states has been processed, it is considered to have been "visited", and is not processed again.

To illustrate the merging process we use the example in Figure 5.10. At the top of the figure are shown two state machines that implement parallel branches. On the bottom of the figure is shown the state machine that results from the merger of the two machines. States s6 of FSM 1 and s11 of FSM 2 are used for handshaking and are eliminated when the machines are merged. The last states of the merged machine have unconditional transitions (not shown) to the state that implements the actions following the parallel branch. State *m1* of the merged machine has only two transitions, with conditions identical to those of *s1* of FSM 2. Any outputs associated with *s1* of FSM 1 are asserted during both transitions of *m1*, that is, unconditionally. Because the first two transitions of FSM 1 are unconditional, the merged machine has the same number of states and transitions as FSM 2 alone up

[5.11] $State_j$ branches to $nextstate_j$ under conditions $cond_j$.

to the third level. At the third level however, there is an explosion of transitions. This is due to the combination of a two way transition from *s3* of FSM 1, along with a four way transition from *s6* of FSM 2. To represent all combinations of input conditions, *m6* must have eight transitions. The merged machine has eighteen states whereas the two separate machines have seventeen states altogether. The elimination of three handshaking signals may compensate for the increase of one state. In the current implementation, however, the multiple state implementation would be chosen because it has fewer states.

5.3.2 Time Constraint Satisfaction

In this section we consider two factors that complicate design with timing constraints: internal loops and conditional branches. *Internal loops* create problems because it is difficult to determine the number of iterations and therefore the effect on timing constraints. *Conditional branches* cause problems when paths are of different lengths. A synthesis technique that is often used is to schedule events into time slots. Events succeeding conditional branches are scheduled to slots that follow the longest branch of the conditional. Any "empty" slots that result along the shorter conditional paths can be skipped over by the controller. If a design system accepts minimum time constraints, however, it is not apparent whether or not the empty control slots are performing a timing function. Empty slots cannot be skipped over for fear of violating time constraints. This situation can lead to inefficient control design with many extra states.

Minimum and maximum constraints are expressed in the DDS as arcs between two points in the timing graph labelled with a time and one of the relations: less than, less than or equal to, greater than, or greater than or equal to. For each minimum and maximum constraint, CONSPEC first:

1. Finds all paths from the head to the tail of each constraint arc through the graph.

2. Determines the lengths, including the contribution of loops, of each path.

3. Notes any timing violation.

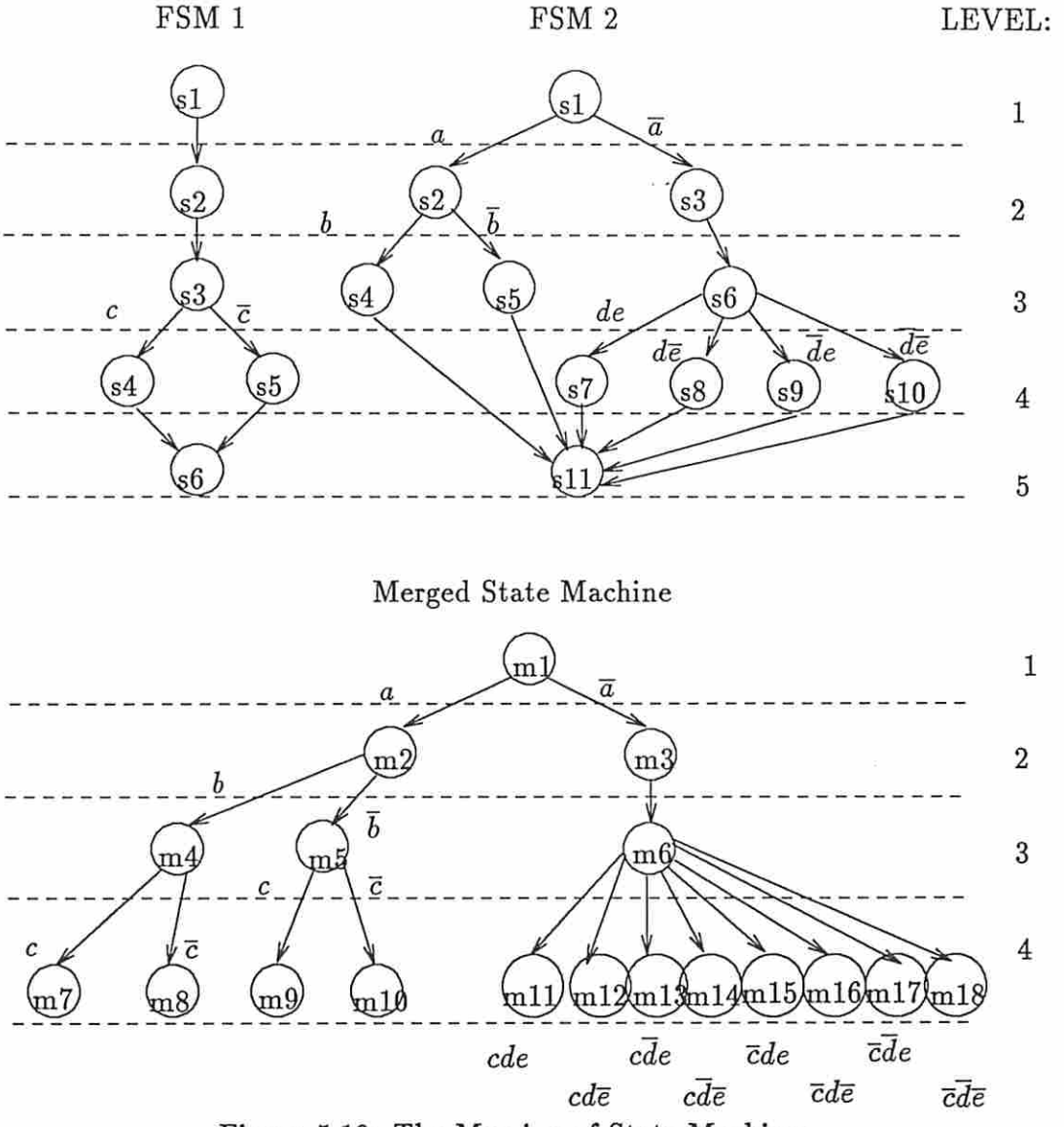


Figure 5.10: The Merging of State Machines

The paths are found by traversing all branches between the states at the head and tail of the constraint. These states are identified by checking the links between points of the DDS timing graph and states. Each path from the first state to the last is represented as a list of the states on the path. Loops are not traversed in finding paths.

After each path contained within a constraint is found, its length is determined. The length without loops is simply given by the list length of the path representation minus one. For minimum constraints, loop lengths are not calculated unless some path violates the constraint because loop iterations can only increase the length of the path and not decrease it. For maximum constraints, however, the loop length is always taken into account as described in Section 5.3.2.2.

Loop iterations are governed by three parameters: the initial value, *init*, the incrementing value that is added, *incr*, and the final value against which a comparison is made before another iteration is executed, *final*.[5.12] The addition of *incr* and comparison with *final* may be done at different times and give different numbers of loop iterations, depending on the type of loop. The loop exit may be at either the top or bottom of the loop.

Given a high level language loop construct such as a Do or For loop, it is possible to define a DDS *template* that can be recognized by CONSPEC in a specification and used to determine the number of loop iterations. The timing graph represents the loop by begin- and end-loop points. The exit from the loop is a conditional transition governed by a synchronous predicate. This conditional transition will exit the loop, and will be from either the first state of the loop or the last. The incrementing and checking operations can be defined completely in the DDS data flow and timing graphs.

Figure 5.11 is used to illustrate the process. *Test* is a synchronous predicate governing looping behavior in the timing graph. During the final state of the loop, *sf*, a transition is made out of the loop if $test = 0$, otherwise the loop is re-entered. CONSPEC detects the presence of a loop and determines that the predicate *test* limits the number of iterations. It then examines the DDS specification and looks

[5.12] CONSPEC does not accept specifications with go to statements. Only structured programming constructs are allowed.

for a match between the data flow graph segment surrounding *test* and the loop-templates. The data flow graph of Figure 5.11 shows the template for a do-loop. To match this template, *test* must be the output of an operation of type less-than-or-equal, the inputs of which must be related to operations of the types shown in the template.[5.13] When a template-match is found, CONSPEC knows which data flow values of the specification correspond to the loop parameters init, incr, and final. It can then use a formula associated with each template to determine a relation between these that defines the number of loop iterations. This is multiplied by the length of the loop path and added to the length of the total path to give the total length of the path including loop iterations.[5.14]

If all specifications are translated from a high level language, and if templates are defined for every looping construct of that language, a match can always be made. In this way CONSPEC determines if a loop is fixed or not, and if so, the number of iterations. When the initialization, increment, and final values governing loop behavior are specified as constants in the data flow graph, the number of loop iterations can be determined. If any of the three values are variables input from the environment, however, the number cannot be determined at design time. Also if a template match cannot be made CONSPEC assumes that the number of iterations is unknown.

In the next two sections we describe the methods used by CONSPEC to satisfy minimum and maximum time constraints.

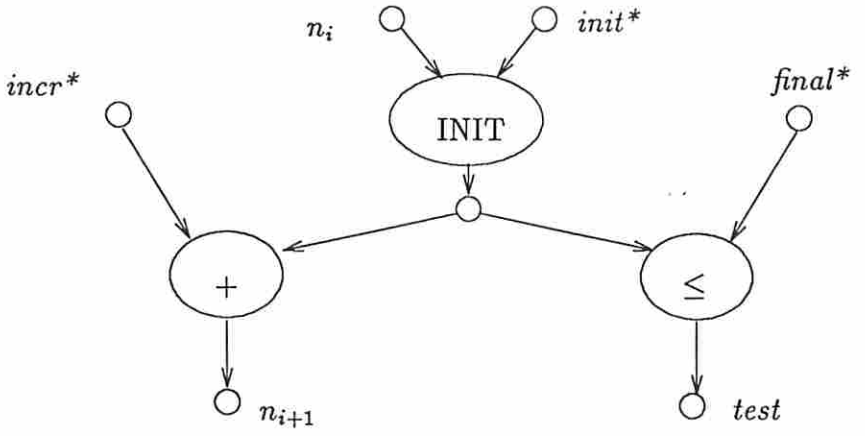
5.3.2.1 Minimum Time Constraints

We do not consider the impact on performance of adding more states to the controller to satisfy minimum time constraints. The state table is derived for a user specified clock period. If the size of the state machine makes it impossible to achieve that speed, then the specification may require rewriting or the clock period must be adjusted.

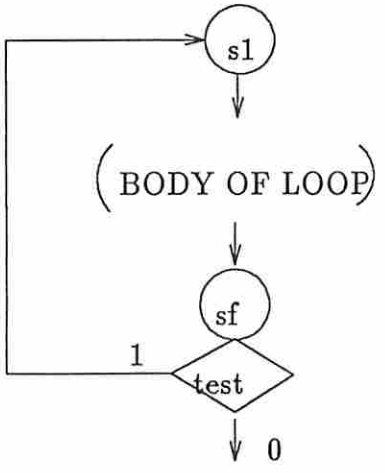
The first step in time constraint satisfaction is the enumeration of all possible sequences of state transitions from the first state of the constraint to the last state.

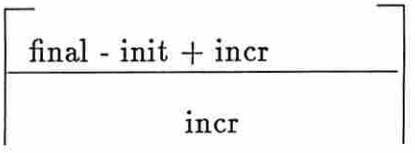
[5.13] Operations are named by appending a number to the operation type, so determination of the type of a particular operation is very straightforward.

[5.14] Nested loops are not currently handled.



DATA FLOW GRAPH TEMPLATE





STATE TABLE NUMBER OF ITERATIONS

Figure 5.11: DDS Do-Loop Template

Each state sequence is called a path. If all the state transitions are unconditional, there will be a single path. The existence of nested conditional forks will result in several paths within the range of a single time constraint.[5.15] Because inequalities in path length may arise from inequalities in conditional branch length, the length of each path must be calculated to determine if the minimum constraint is met. If there are any internal loops with indeterminate length we assume that the minimum number of iterations occurs, which is the worst case. That means that the presence of the loop does not add any time to the path, for purposes of minimum time constraint satisfaction. This is adequate to guarantee the minimum timing.

Any path that does not meet the time constraint is termed a **short path**, while any that meets or exceeds the requirement is termed a **long path**. CONSPEC lengthens short paths to meet the constraint by adding extra states, called **timing states**; these states have no outputs and their next state transitions are unconditional.

If there is a single path through the constraint, CONSPEC satisfies violated minimum constraints by inserting a number of timing states equal to the time constraint minus the length of the path. These are placed between the last two states of the path's state sequence. It is obvious for this trivial case that the constraint is satisfied and that the minimum number of states is added.

If there is more than one path, the absolute minimum number of states that could be added to satisfy the constraint is equal to the time constraint minus the length of the shortest path. If we insert this number of timing states immediately before the last state within the constraint such that all paths are lengthened equally, then it is obvious that the resulting paths will all meet or exceed the requirement. This method is generally not satisfactory, however, since any path longer than the shortest path is lengthened more than required, thereby degrading performance.

The minimum constraint satisfaction method that CONSPEC uses lengthens short paths by placing timing states on the short branches of conditional forks,

[5.15] Minimum time constraints that include parallel branches nested within their range are not currently supported, unless the parallel sequences can be merged into a single serial sequence prior to being input into CONSPEC.

thereby reducing or eliminating differences in branch and path length. This is illustrated by Example 1 in Figure 5.12,[5.16] which shows on the left a state diagram of a process constrained to a minimum time of eight clock periods from S1 to S8. The constraint can be satisfied by adding timing states in the positions indicated by double circles on the state diagram. A design such as this results when operations are scheduled according to data readiness and time constraints as in [NT86]. The CONSPEC approach differs from this approach in two ways. First, timing states are shared between the branches of each conditional fork; fewer states are therefore added to satisfy the constraint. This is possible because no outputs are produced during timing states and because transitions from timing states are always unconditional. The output of CONSPEC for Example 1 is shown by the state diagram on the right of Figure 5.12. The sharing of timing states for this example saves a total of four states compared to the design on the left. Second, when the critical path (i.e. longest path) exceeds the time constraint in length, CONSPEC adds fewer states and lengthens fewer paths beyond the minimum required. Example 2 in Figure 5.13 illustrates this situation.

A minimum time constraint of eight clock periods is imposed between S1 and S13 in Example 2. The critical path is nine clock periods long, but none of the other seven paths meets the constraint. We could satisfy the constraint by making all branches of each of the four conditionals the same length, through the addition of states in the positions indicated by double circles on the state diagram. This is the design that results from scheduling operations according to data readiness and time constraints. Since the critical path length exceeds the minimum timing requirement, however, more delay is added than necessary to meet the constraint. This degrades the performance and also results in a larger than necessary number of control states. It is not a trivial matter to "skip over empty control slots" for a design like this, since some are required to meet the constraint and it is not obvious which ones and how many should remain.

If the critical path exceeds the time constraint it may be impossible to lengthen short paths without at the same time lengthening other paths beyond the minimum required. This happens when all branches that constitute the short path also lie on some long path, as is the case for Example 2. CONSPEC uses heuristics to place

[5.16] Input conditions for state transitions are not shown in the figures in this section.

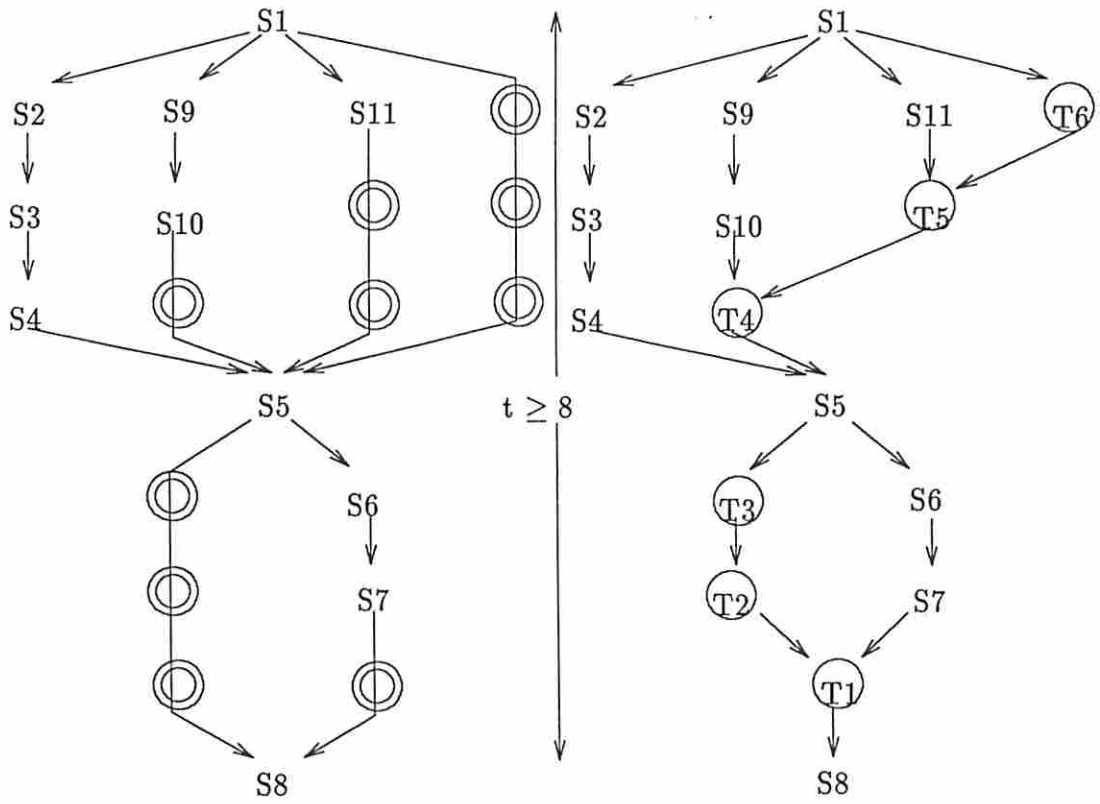


Figure 5.12: Minimum Time Constraint Satisfaction, Example 1

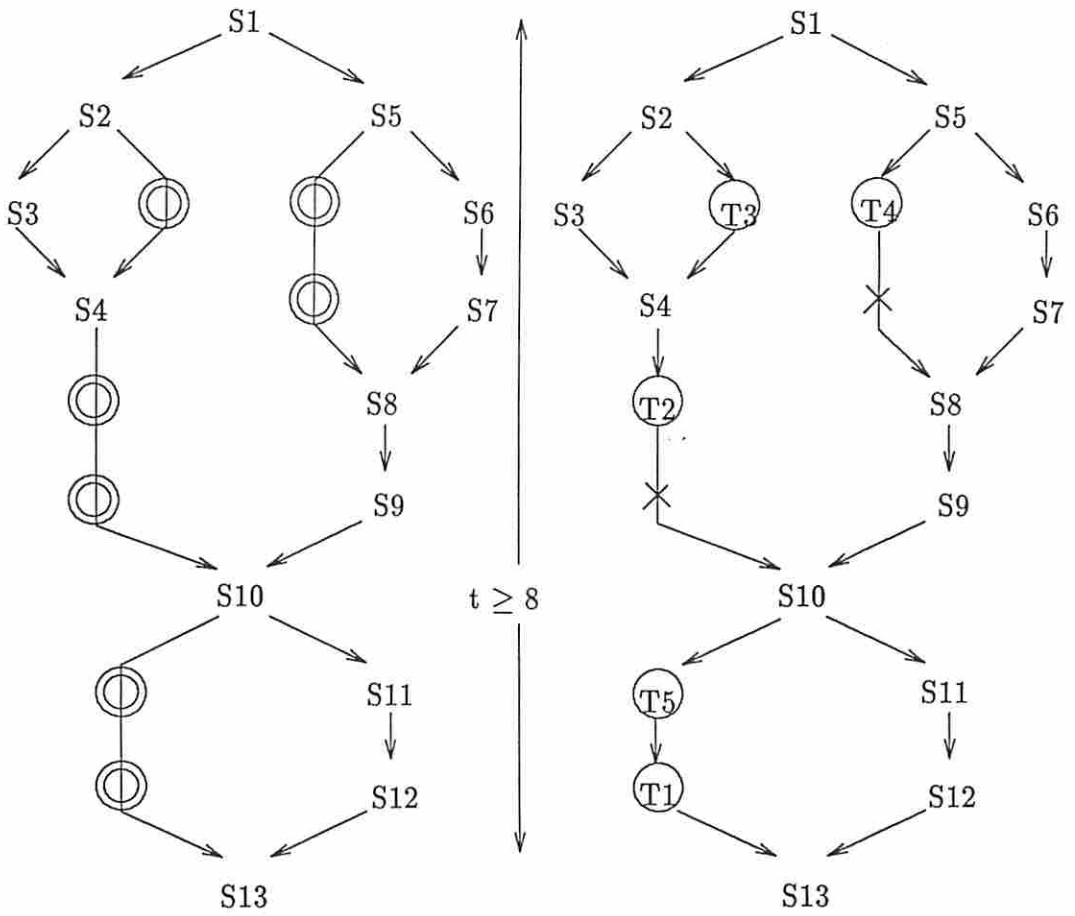


Figure 5.13: Minimum Time Constraint Satisfaction, Example 2

timing states in locations where the impact on long paths is minimized. The state diagram on the right side of the figure shows the solution created for Example 2. The time constraint has been satisfied without adding two of the double circled states. The omission of these states not only reduces the total number of control states, but results in six shorter paths. An alternate design would be to omit T5 and place timing states at the locations marked X; this would result in one extra timing state, however. In addition to minimizing the impact on long paths, heuristics are used to reduce the total number of timing states added.

The constraint is satisfied in two phases. During Phase I, states are added to meet the time constraint on short paths only when this can be done without inadvertently lengthening some other path beyond the minimum requirement. If the length of the critical path through the constraint is less than or equal to the minimum time constraint, Phase I is guaranteed to satisfy the constraint and uses

the minimum number of timing states possible without lengthening any paths beyond the required length. If the critical path through the constraint is greater than the minimum time constraint, some short paths will remain after Phase I. During Phase II heuristics are used to place timing states in locations where the the impact on the overall system performance and controller area is minimized. Phase II in combination with Phase I is guaranteed to satisfy the constraint for these cases. The two phases are described in detail in the following paragraphs.

5.3.2.1.1 Phase I During Phase I CONSPEC examines each branch of each conditional fork and adds the number of timing states required by the longest path passing through the branch. If that branch lies on a path that meets or exceeds the constraint, therefore, no timing states are added. Path lengths are updated after each addition. CONSPEC minimizes the number of timing states by sharing them between the different branches of the same conditional fork, up to the number required for each branch.

We call the state at which conditional branches terminate a merge state. Each branch of a conditional fork is represented by its last state transition or merge-transition, [S,MS], where S is the last state of the branch and MS is the merge state. Any timing states added to a conditional branch are inserted between the states of the corresponding merge-transition. Merge states are processed in order from the latest, which is closest to the final state of the constraint, to the earliest. The ordering between merge states on different branches of the same conditional fork is arbitrary. This defines a partial order on the merge states. For Example 2, S13 is the latest merge state and S10 is next. S8 and S4 follow S10 but neither is considered later than the other so they can occur in any order relative to each other. An outline of Phase I is given below:

- For each merge state, S_i, latest to earliest:
    1. For each $branch_{ij}$ converging on S_i, let N_{ij} be the time constraint minus the length of the longest path passing through.
    2. Let $M_i = \max N_{ij}$ for all j.
    3. If $M_i > 0$ then add a sequence of M_i timing states preceding S_i, with a transition to the N_{ij}'th timing state from $branch_{ij}$, when $N_{ij} > 0$.

4. Update lengths of all paths affected.

If the critical path length equals the minimum time constraint, Phase I makes all branches of each conditional the same length and thereby increases the length of each path to that of the critical path.[5.17] If the critical path length *CP* is less than the required minimum *TC*, then in addition to equalizing the length of all conditional branches, during Phase I *TC* - *CP* timing states are inserted before the latest merge state such that all paths are lengthened equally. For Example 1, $TC - CP = 1$, so T1 is added immediately before S8 such that all paths are lengthened by one. In this way the time constraint is satisfied. This can be demonstrated by examining the effects of Phase I in terms of branch lengths at each conditional fork.

First we observe that for each path passing through the longest branch of a conditional fork, there exists a corresponding path that is identical except that it follows a shorter branch of the same fork. The path on the shorter branch therefore requires the same number of timing states as the path through the longer branch, plus the difference in length between the branches. From this observation it follows that at each merge state S_i, the longest path passing through $branch_{ij}$ with branch length L_{ij} will require:

$$N_{ij} = L_{ix} - L_{ij} + N_{ix}$$

timing states, where $branch_{ix}$ is the longest branch converging on S_i, L_{ix} is its length, and N_{ix} is the number of timing states required by the longest path through $branch_{ix}$.

CONSPEC specifications must have properly nested branches and structured program control; for this reason all paths passing through an earlier merge state will also pass through any later merge state. For Example 2, all paths passing through S8 will also pass through S13. Consequently every path bound by the constraint, including the critical path, passes through the latest merge state, S_l. This state is processed during the first iteration of Phase I. It is obvious that the critical path will pass through the longest branch converging on S_l, which we will call $branch_{lx}$. N_{lx} therefore equals *TC* - *CP*. As shown by the equation above, the

[5.17] Because there may be nested conditional forks we define the length of a branch to be the length of the longest path through the branch.

longest path on every shorter $branch_{lj}$ will require that many states plus a number equal to the difference in branch length. If $TC > CP$, therefore, a number of states equal to TC - CP are added preceding the latest merge state, such that all branches are lengthened equally. In addition, $L_{lx} - L_{lj}$ timing states are added to each $branch_{lj}$, which equalizes the lengths of all branches of the conditional.

For Example 1 on page 152, S_l is S8. The longest branch has merge transition [S7,S8], and its length, L_{lx}, is two states more than the length of the shorter branch, L_{lj}, with merge transition [S5,S8]. The time constraint is one more than the critical path, so the number of states required by the longer path, N_{lx}, equals one. The shorter branch therefore requires $2 + 1 = 3$ states. T1 is added to both branches and thereby to all paths. The lengths of the two branches are equalized with the addition of T2 and T3.

Therefore for cases in which the critical path equals or is less than the time constraint, after the first iteration of Phase I the critical path meets the time constraint and the branch lengths of the latest conditional fork have been equalized. To meet the time constraint for all paths the branch lengths of all other conditional forks must also be equalized.

The length of the longest path through each branch converging on S_l now equals the time constraint. Since at least one of these branches comes from the next earlier merge state, S_n, it follows that at least one path that meets the time constraint passes through S_n. This path will pass through the longest branch converging on S_n, which consequently requires no timing states. Because $N_{nx} = 0$, the longest path passing through every other $branch_{nj}$ with length L_{nj} requires $N_{nj} = L_{nx} - L_{nj}$ timing states, where L_{nx} is the length of the longest branch. The addition of these states equalizes the branch lengths to L_{nx} and lengthens the longest path through each branch to meet the time constraint.

To illustrate this we return again to Example 1 on page 152. For merge state S5 the longest branch is the one with merge-transition [S4,S5]. The longest path through this branch requires no timing states after the addition of T1, T2, and T3 at the latest merge state. The shortest branch of the fork is three states shorter than the longest branch. For this reason, the largest N_{nj} of the three short branches equals three. A total of three states are therefore inserted before S5. T4 is shared between all three short branches and in addition T5 is shared between the two

shorter branches. T6 alone is not shared, since only the shortest branch requires three timing states.

The same observations on the processing of S_n hold for each subsequent merge state. After all merge states have been processed and Phase I terminates, each conditional fork has branches with equal length; consequently all paths have been lengthened to the critical path length. Since the critical path length was increased to the required minimum time, it follows that all paths meet the time constraint. Because the number of states added at each conditional fork is equal to the largest N_{ij} of all the branches, which share the timing states, the minimum number of states has been added.

We next analyze Phase I for cases in which the critical path is longer than the time constraint. For the first merge state, S_l, the longest branch is on the critical path; the value of N_{lx}, given by $TC - CP$, is negative and therefore no timing states are added to the longest branch. Timing states are added on the other branches only when $L_{lx} - L_{lj} + N_{lx}$ is greater than zero. This occurs when the difference in branch length, $L_{lx} - L_{lj}$, is larger than the absolute value of N_{lx}. The same observation holds for the branches of other conditional forks. The lengths of the branches are not equalized, because that would lengthen all paths beyond the minimum required.

We illustrate this using Example 2 on page 153. A minimum time constraint of eight clock periods is imposed between S1 and S13. Table 5.1 lists the paths in order of decreasing length. The *Add* column indicates the total number of timing states that must be added to meet the time constraint for each path. The *Branches* column indicates the component branches of each path represented by their merge transitions. There are a total of eight paths between S1 and S13 with lengths varying between 4 and 9 state periods. Only the path following the right branches through the diagram is long enough, at nine clock periods.

The state diagram on the right side of Figure 5.13 shows the solution created by CONSPEC. During Phase I timing states T1, T2, T3, and T4 are added as shown. Merge state S13 is processed first; its two converging branches are represented by the merge-transitions [S10,S13] and [S12,S13]. Long Path 1 has merge-transition [S12,S13] so no timing states can be added to the right branch. On the left branch, the longest path with merge-transition [S10,S13] requires one extra clock period,

Path	State Sequence	Branches	Length	Add
1	s1,s5,s6,s7,s8,s9,s10,s11,s12,s13	[s7,s8],[s9,s10],[s12,s13]	9	0
2	s1,s5,s6,s7,s8,s9,s10,s13	[s7,s8],[s9,s10],[s10,s13]	7	1
3	s1,s5,s8,s9,s10,s11,s12,s13	[s5,s8],[s9,s10],[s12,s13]	7	1
4	s1,s2,s3,s4,s10,s11,s12,s13	[s3,s4],[s4,s10],[s12,s13]	7	1
5	s1,s2,s4,s10,s11,s12,s13	[s2,s4],[s4,s10],[s12,s13]	6	2
6	s1,s2,s3,s4,s10,s13	[s3,s4],[s4,s10],[s10,s13]	5	3
7	s1,s5,s8,s9,s10,s13	[s5,s8],[s9,s10],[s10,s13]	5	3
8	s1,s2,s4,s10,s13	[s2,s4],[s4,s10],[s10,s13]	4	4

Table 5.1: Example 2: Paths Through Time Constraint

so T1 is added in the location shown.[5.18] Merge state S10 is processed next. The longest path passing through the left branch (merge-transition [S4,S10]) requires one extra clock period. Long Path 1 passes through the right branch, so no states can be added there, therefore T2 is added as shown between S4 and S10.

At merge state S4, the longest path with merge-transition [S2,S4] requires one extra period, the left branch has long paths passing through, so T3 is added to the right branch. The case is similar at S8, where the right branch can take no timing states. The longest path passing through [S5,S8] requires one more clock period to meet the constraint, so T4 is placed between S8 and S5. Once these states have been added the longest path through each branch meets the time constraint, but the three shortest paths still require timing states.

5.3.2.1.2 Phase II Phase II is guaranteed to satisfy the time constraint for any short paths left after the completion of Phase I. Short paths will remain only for specifications where the critical path exceeds the time constraint. Four heuristics are used to place timing states so as to reduce the impact on performance and area. Two are performance measures; in order of importance they are the length and the number of long paths affected by the addition of timing states. The length of the affected paths is considered more significant than the number of affected paths. This focuses primarily on the the worst case system performance rather than the

[5.18] Timing states are numbered in the same order they are added, so T5, for example, has not been added at this stage of processing.

average performance.[5.19] As an example, CONSPEC considers it less damaging to lengthen two long paths than to lengthen a single path that is even longer. Within the restrictions imposed by the performance constraints, two steps are taken to minimize the total number of timing states added (and therefore controller area). First, timing states are placed on a branch common to the greatest number of short paths. Second, timing states are shared between branches of the same conditional fork up to the number required by each branch, just as is done during Phase I.

Short paths are enumerated and processed from the longest, requiring the fewest timing states, to the shortest, requiring the most. For each short path a timing state location is chosen by examining each branch on the path. Paths are represented by a list of merge-transitions, one for each branch that the path passes through, plus path length. The lengths of all paths that will be affected by the addition of timing states on the chosen branch are immediately updated, but timing states are not added until all locations are chosen, so that they can be shared between the branches of the same conditional fork. An outline of Phase II is given below:

1. For each short path, longest to shortest: **Choose locations**
   (a) For each merge-transition find the longest path that passes through. Retain merge-transition(s) for which this length is the smallest.
   (b) For each remaining merge-transition determine the number of long paths that pass through. Retain merge-transition(s) for which this value is the lowest.
   (c) For each remaining merge-transition determine the number of short paths that pass through. Retain merge-transition(s) for which this value is the highest.
   (d) If more than one merge-transition left, choose first and record choice.
   (e) Do not add timing states to path yet, but update lengths of all paths that will be affected by addition.
2. At each merge state S_i add timing states recorded during Step 1a:

[5.19] An alternate heuristic could be used that considers the number of long paths affected before their relative length.

(a) Let M_i be the largest number of timing states recorded at Step 1d for any transition to S_i.

(b) Add M_i timing states immediately before S_i, sharing between branches up to the number associated with the corresponding merge-transition.

Phase II is guaranteed to satisfy the time constraint, since all short paths are enumerated and the number of states required by each is added to one path at a time, updating path lengths after each addition. The locations of timing states are chosen such that impact on performance is minimized based on path length and the number of long paths affected. Within the restrictions imposed by the performance constraints, the number of timing states is minimized by adding them where the greatest number of short paths are lengthened, and by sharing them between branches of the same conditional fork.

Although at each step the timing states are added at a single location, by processing the paths in order from longest to shortest, the timing states on the shorter paths get distributed between different branches. This is because longer paths pass through many of the same branches as shorter paths. The addition of timing states on branches in common with the shorter paths contributes to time constraint satisfaction on those as well.

In general, the timing states added at a single location will never lengthen a branch beyond the length of the longest branch of the same conditional fork. This can be seen by assuming that such a lengthening does occur, and showing that the assumption leads to a contradiction. Using Figure 5.14 to illustrate, assume that a path passing through the left branch requires two states to meet the time constraint, and that the timing states are added on this branch. This action lengthens the branch beyond the length of the longest branch of the fork, which we claim CONSPEC does not do. But note there is a second path, identical to the first, except that the branch on the right is followed instead of the left. Since the branch on the right is one state longer than the left branch, it follows that the second path must require one timing state to meet the constraint. This contradicts the fact that, because the second path is longer than the first and is therefore processed earlier, all necessary timing states have already been added and it therefore requires zero timing states. It is therefore impossible that a short branch will be lengthened beyond the length of the longest branch of the conditional.

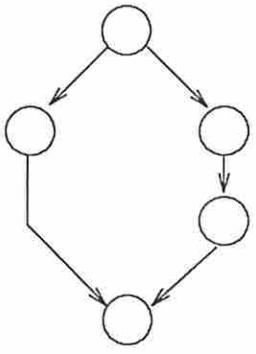


Figure 5.14: Distribution of Timing States during Phase II

We illustrate Phase II using Example 2 on page 153 (table on page 158). Paths 6, 7, and 8 still require one timing state each. Path 6 is processed first. During Phase I, the length of Path 6 was increased to seven by the addition of timing states T1 and T2, so one more is required. At this point, however, every branch on Path 6 is shared with some path that meets the time constraint: [S3,S4] with Path 4, [S4,S10] with Paths 4 and 5, and [S10,S13] with Path 2. The addition of the state necessary to meet the constraint on Path 6 will therefore unavoidably increase the length of some other path beyond the minimum required.

During Step 1(a) merge-transitions where the longest paths pass through are eliminated as locations for timing states. For each merge-transition on Path 6, however, the longest path passing through is eight clock periods, so for this example this criteria does not narrow down our choice. The branches where the fewest long paths will be affected are [S3,S4] and [S10,S13]. Of these two locations [S10,S13] is chosen because the two other short paths, 7 and 8, share the same merge-transition; a single timing state T5 is therefore placed as shown in Figure 5.13. This addition satisfies the time constraint for Paths 6, 7, and 8, thus completing minimum time constraint satisfaction.

If timing states are added at all locations marked by double circles in the state diagram on the left of the figure, all seven short paths are lengthened beyond the minimum. Since CONSPEC omits states in the two locations marked by X in the right diagram, only path 2 is lengthened one more than the minimum. This lengthening was unavoidable to meet the constraint on the three shortest paths. One correct but inferior solution results if T5 is omitted and one timing state is placed at both locations marked X. In that case, the same number of paths would

be lengthened beyond the minimum (path 4 would have been lengthened to 9 clock periods instead of path 2), but one more timing state would be required. Placing the timing state at the location of T5 allows three short branches to share a single timing state.

5.3.2.2 Maximum Time Constraints

The first steps of time constraint satisfaction outlined at the beginning of Section 5.3.2 are enumeration of all paths within the constraint and determination of their length, which must always include loop length. If the number of loop iterations cannot be determined, the maximum length of the path is unknown and CONSPEC cannot guarantee that the constraint will be met. In that case there is no attempt to satisfy the constraint. Processing continues, but the maximum constraint is marked as being potentially violated because of paths with unknown length.

The data path synthesis systems in ADAM satisfy maximum time constraints related to data flow operations. Other maximum time constraints may be imposed on behavior unrelated to the data path, such as bus interactions or external control functions. Because the specification may be constrained by interaction with the environment, sequencing requirements cannot be determined from data dependency alone. CONSPEC therefore assumes that it cannot violate sequencing requirements specified in the DDS timing graph by arbitrarily placing sequential events into the same state, even if the combined lengths of the intervals do not exceed the clock period.

Events in the timing graph are executed as soon as possible given the sequencing requirements in the timing and control graph. Procedure and subroutine calls, for example, do not result in the specification of extra control steps for call and return events but are handled as forms of conditional transition.[5.20] For these reasons, and because module selection is not considered here, there are a limited number of options available to meet maximum constraints.

Nevertheless, two general strategies can be employed that may reduce path lengths, depending on the nature of the particular specification.

[5.20]The logic implementation later chosen for the controller will finalize the manner of effecting such branches.

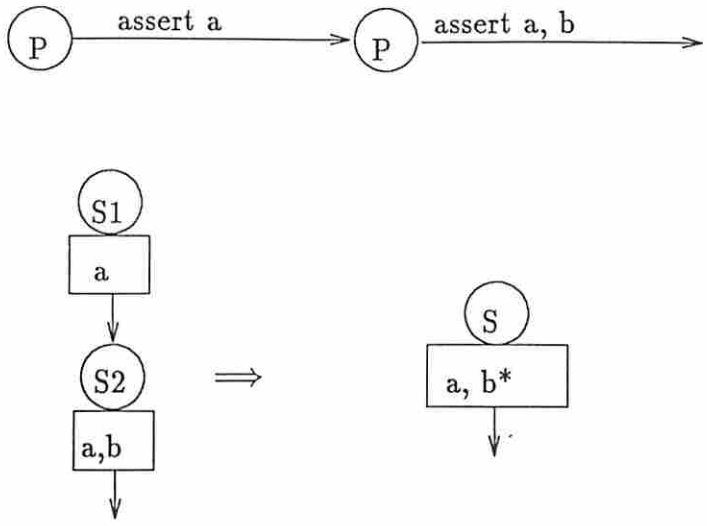


Figure 5.15: Clock Suppression of Outputs

5.3.2.2.1 Early Assertion of Conditional Outputs There are two cases in which CONSPEC can eliminate a state by asserting a conditional output one state earlier.[5.21] The timing graph at the top of Figure 5.15 shows two sequential intervals. Output *a* must be asserted before output *b*. If a=data and b=data ready, for example, then the data must be stable before the ready signal is asserted. The default implementation is shown by the state table fragment on the bottom of the figure. Regardless of the length of the first interval, *s1* is assigned to assert *a* for a full clock period before *b* and *a* are asserted in *s2*. If the interval during which *a* must be asserted alone is short relative to the clock period, however, it is possible to use *clock suppression* to eliminate *s2*, as shown by the state table fragment to the right of the arrow. The star on *b* indicates that it should not be asserted until the second clock phase of state *S*. This guarantees that *a* is asserted for half a clock period before *b*. Clock suppression is a common conditional output strategy discussed in elementary design texts [Com84] that employs gates at the outputs of the state machine. The inputs of the gate are the signal itself (*b*) and the clock signal, possibly connected to an inverter. The lengths of the first and second intervals must be no greater than half a clock period to employ this method.

A somewhat more complex clock suppression strategy not shown in the figure is employed if some outputs associated with the first interval should not be asserted

[5.21] The Mealy model is being used, in which control outputs depend on the values of inputs as well as state.

during the second interval. In that case, clock suppression can be used to assert outputs associated only with the first interval during the first clock phase, and outputs associated only with the second interval during the second clock phase of a single state, while outputs associated with both intervals are asserted during both phases. In that way two short adjacent intervals can be implemented by one state.

The timing graph at the top of Figure 5.16 illustrates the second case in which CONSPEC can eliminate one state by asserting conditional outputs early. During the interval succeeding P an operation is executed, followed by a conditional branch dependent on the results of the operation (the data flow graph would show an AND operation with output *and*). The default implementation created by CONSPEC is shown in the state table fragment below and to the left of the graph. The operation is scheduled during *s1*, the value of *and* is checked during *s2* and a branch made accordingly. Under the right circumstances, however, it is possible to execute the operation, check the results, and perform the branch actions during the same state, as shown by the state diagram fragment to the right of the arrow. If the operator assigned for the implementation is fast, as indicated by a short length on the interval following P compared to the clock period, it can be done without race conditions. As the length of the operation increases and approaches the clock period, however, race conditions can result, therefore this is a less conservative design approach. For this reason the strategy is employed only if the length of the first interval is somewhat less than half a clock period.

In general, this strategy is applicable whenever the interval preceding a conditional branch is short enough, and the branch is made based on a value calculated in the data path. If the value is an input from the environment, however, nothing can be done, since the value of the input may not be valid at the earlier time period and we may violate sequencing requirements.

Using the methods discussed in this section, we may be able to reduce path length, depending on the particular specification. There is no guarantee, however, that maximum constraints can be met in this way.

5.3.2.2.2 Reduction of Clock Period The reduction of the clock period is a more global strategy that has the potential to reduce path length. Because we

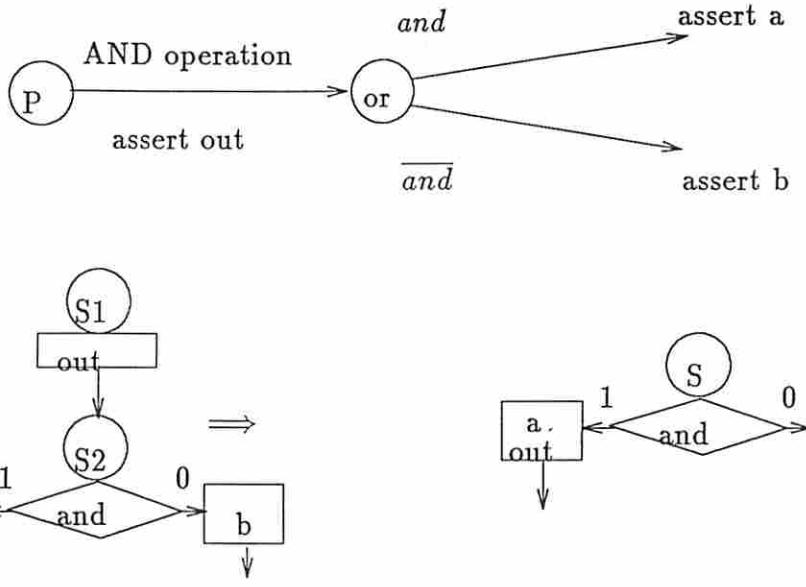


Figure 5.16: Early Assertion of Conditional Outputs

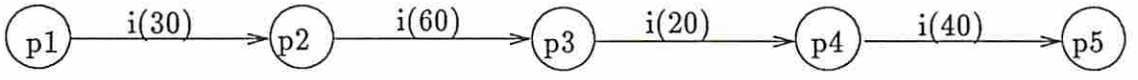


Figure 5.17: Reduction of the Clock Period

are using a synchronous strategy, the behavior is executed one clock period at a time. Since not all operators take the same time, and because time intervals in the specification may be of arbitrary lengths compared to the chosen clock period, short operations may complete sooner than one clock period, and intervals shorter than a clock period will be assigned one state. This results in wasted time. An example timing interval is shown in Figure 5.17; below is listed the number of clock cycles or states that will result, along with the total time, given decreasing clock periods.

Clock Period	Clock Cycles	Total Time
60 ns	4	240 ns
40 ns	5	200 ns
30 ns	6	180 ns
20 ns	8	160 ns
10 ns	15	150 ns

Decreasing the clock period below 10ns will not result in greater time savings, as this is the greatest common divisor (GCD). In general, the clock period resulting

in the greatest speed for an arbitrary timing graph will be the GCD. Unfortunately, not only is the solution of this problem computationally expensive, it is also not generally useful, since for an arbitrary graph the GCD is likely to equal one unit of the clock period.[5.22] For our purposes as well, we are interested in meeting constraints rather than in finding the fastest implementation. This is particularly true because as the clock period decreases, the number of states and the demands on controller performance (speed) increase. Thus we are trading speed for area, unlike the strategies discussed above, in which states were eliminated to improve speed. A preferable strategy would be to decrease the clock period only enough to meet the constraints. There is no guarantee, nevertheless, that reduction of the clock period will find a value at which all maximum time constraints are met. That depends again on the particular timing graph of the specification.

This thesis does not address clocking scheme synthesis, which is a different problem.[5.23] The current approach to the reduction of the clock period is as follows:

1. Propose reduction of the clock period to the user and provide them with the following information:

   (a) The current clock period.

   (b) A list of interval lengths.

2. If the user suggests a clock period, redo design with new clock period
   Else report maximum time constraint violation and complete current design.

A third plan could be added to CONSPEC that would determine the greatest common divisor (GCD) and set the new clock period to this value. The precondition for this plan should be a procedure that requests the user to evaluate the potential usefulness of applying this method. A lower bound for the clock period length could be specified so that if the GCD found was less than the lower bound, the plan would fail and the clock period would not be reset.

[5.22] This would happen if the 20ns interval above were 23ns, for example.

[5.23] See reference [PP85]. There is also some unpublished work by Zahir and Fichtner of ETH, Zurich, dealing with a similar problem, presented at the High-Level Synthesis Workshop in Kennebunkport, Maine, October 1989.

5.4 Examples

5.4.1 Partial UNIBUS Arbiter

Figure 5.18 shows the DDS timing graph specification of the partial UNIBUS arbiter at the top, and the implementation derived by CONSPEC at the bottom. The transitions of the state diagram are annotated by the name of the interval whose behavior the transition is implementing. We first list the values of the various conditions calculated for each of the three intervals that reflect the effects of synchronous and asynchronous signals, and then discuss the derivation process:

1. Synchronous Signals
   i1 = [ ]; i2 = [lss]; i3 = [not(lss)]

2. Asynchronous Signals: values *not* resulting in branch
   11 = [ ]; i2 = [not(sack)]; i3 = [ ]

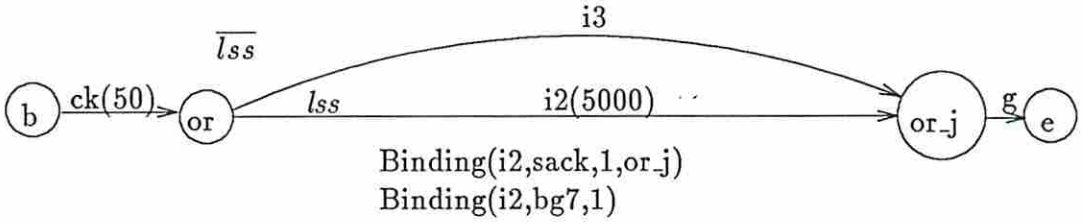
3. Asynchronous Signals: values that *will* result in branch.
   i1 = [ ]; i2 = [sack]; i3 = [ ]

4. Asynchronous + Synchronous Signals $\Rightarrow$ Cond (1) $\wedge$ Cond (2)
   i1 = [ ]; i2 = [lss $\wedge$ not(sack)]; i3 = [not(lss)]

5. Asynchronous + Synchronous Signals $\Rightarrow$ Cond (1) $\wedge$ Cond (3)
   i1 =[ ]; i2 = [lss $\wedge$ sack]; i3 = [not(lss)]

The first interval has bound to it a data path operation, so its length is equal to the data path clock period, 50ns. This is implemented by a single transition of *s1* to *s2* under Condition 4 for *i1*, which is [ ], or nil. This is equivalent to an unconditional transition.

At the or-point the value of the synchronous input *lss* is checked. If true, branch *i2* is followed. This interval has a length of 5000ns specified, which requires ten state periods, implemented using a loop. Because of the check on *lss*, a second state has to be introduced for the loop, to avoid rechecking the value of the synchronous input. The transitions also depend on the value of an asynchronous input *sack* (selection acknowledge) bound to *i2*. *S2* has a transition to *s1* under Condition 4 for *i3*, which is [not(lss)]. This transition implements interval *i3*, and is not dependent



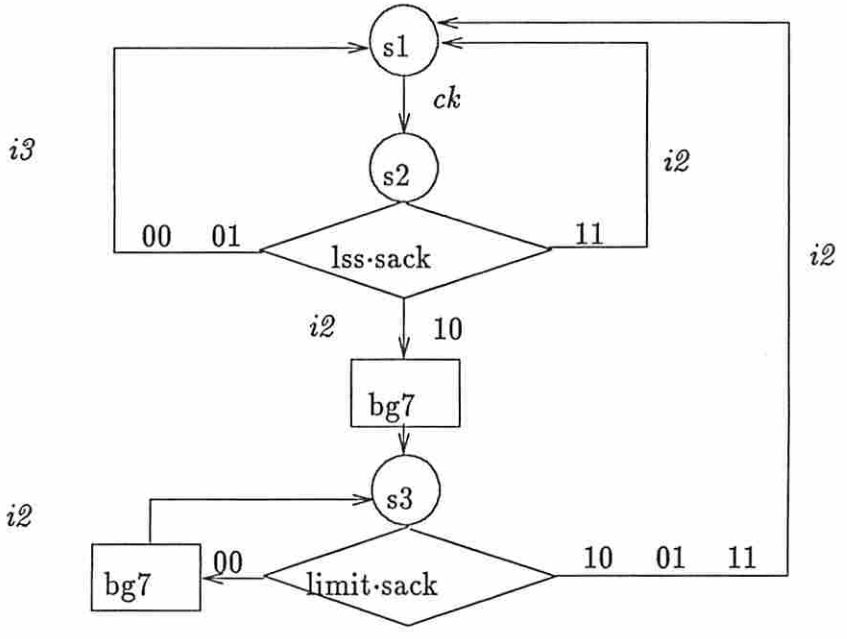


Figure 5.18: Control Synthesis of the UNIBUS Arbiter

on *sack* since there is no asynchronous predicate bound to the interval. Therefore the input conditions for the transition are: $\overline{lss} \cdot sack$ and $\overline{lss} \cdot \overline{sack}$. In the state table printed out by CONSPEC, the actual value given for *sack* would be x, or don't care, for this transition.

S2 also has a transition to the looping state *s3* under Condition 4 of *i2*, which is [lss $\wedge$ not(sack)]. Additionally, *S2* will have a single transition for the asynchronous predicate bound to *i2*, corresponding to the assertion of the predicate. This transition to *s1* is performed under Condition 5 for *i2*, [lss $\wedge$ sack], that is, when lss=1 and the predicate is asserted. The transitions of *s3* are to itself, under Condition 2 = [not(sack)] plus the condition not(limit), that is, when the asynchronous predicate is not asserted and the limit on the number of iterations has not been reached. When either the asynchronous predicate has been asserted or *limit* has been reached a transition is made from *s3* to the same state, *s1*. The output *bg7* is asserted by *s2* and *s3* only so long as *sack* is not asserted and *limit* has not been reached.

5.4.2 Frame Counter

The synthesis process will be described for the frame counter example given in Section 3.4 to illustrate synchronous descriptions. See Figure 3.20 on page 75 for the DDS representation of the behavior.

This example has only one data path operation, which increments the record count. Therefore, it is not necessary to call MAHA to schedule operations. Module set choice must consider only the time constraint of $tc \leq 7$. The cheapest module with the appropriate behavior and performance is chosen. Because this happens to be an incrementing register, it is not necessary to allocate a register to hold count.

Next, the data path control signals must be incorporated into the DDS graph. The increment control signal must be asserted beginning after the fourth check on D's value. This is clear from the specification, from the binding of the operation to a specific interval. The duration of the control signal is, however, not specified, and must be determined by comparing the clock period of the specification to the characteristics of the chosen operator. In this case, the timing requirement for

assertion of the register's control signal fits within the time limit of a single clock phase in our specification. This means that modification of the graph is minor, involving the insertion of a conditional output after the fourth check on the value of D. If some other operator were involved that required assertion of the control signal over multiple clock periods, greater modification would have been required due to the loop. The form of the DDS graph after incorporating the control signal *FE* (for Frame End) is shown in Figure 5.6.

Next the timing and sequencing graph is traversed, starting at the first point, which is a begin-loop point. This point defines the beginning of the first state in the diagram. State names are constructed using the letter *s* appended to a number or letter that uniquely identifies the state. Intervals ending at an end-loop point indicate a state transition to the first state of the loop. Begin- and end-loop points are identified by a unique index, so this index is used to identify the proper state transition. In this particular example, because there is only one loop, no index is required. For each state the system must consider two elements: next state transitions and outputs.

There are no bindings defined on the interval *hi* following point *b*. This means that there are no unconditional outputs. The succeeding point is an exclusive-or branch, which signals a conditional state transition based on the value of the inputs *C* and *D*. When the value of *C* is logic level zero, the state transition is a loop back to the first state of the loop, since the interval *lo* is directed to the end-loop point *e*.[5.24] There are no conditional outputs associated with this transition because there are no bindings defined on interval *lo*. When the value of *C* is logic level one, the next state transition also depends on the value of a second input signal, *D*, specified as a predicate. Subsequent points on both of these branches are *p* points, which indicate new states. There are no conditional outputs associated with the transition to either of these points, as noted by the absence of bindings defined on any of the intervals. Through this process we have identified the outputs (none) and state transitions for the first state.

[5.24]The presence of the or-join point *or_j3* indicates that several transitions to the end-loop point exist, since the *b* point cannot have several incoming arcs. It does not indicate a new state. This can be detected by the type of out-arc from the *or_j* point, which is not a clock or interval arc, but a *null* arc.

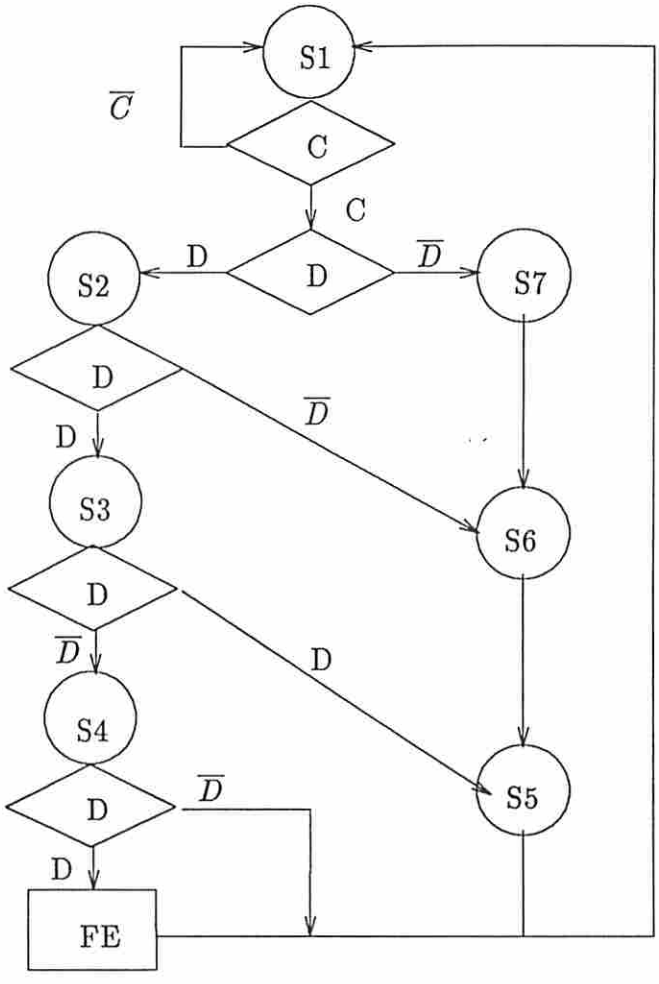


Figure 5.19: State Diagram

The conversion of the timing graph progresses in an identical manner for the subgraph following point *p1*, which is shown on the left in Figure 5.6. This subgraph continues with alternating *or* and *p* points, indicating a sequence of conditional state transitions as described above. None of the states have outputs until we reach the state associated with interval *lo8*, which has a carrier-value binding indicating that a conditional output,*FE*, should be asserted when the input signal *D* is at logic level one during that state. Conversely, the subgraph that proceeds from point *p2* consists of a sequence of *or_j* points, which signifies a sequence of unconditional state transitions. There are no bindings defined on any of the intervals, meaning that there are no outputs asserted during any of the states in this sequence. This analysis leads to the state diagram of Figure 5.19.

5.4.3 Multi-bus Slave

A simple example synthesized by ISYN in [NT86] was chosen to illustrate the method. This is a partial description of the Multi-bus slave interface. In more complex examples there is less correspondence between the DDS graph and the state diagram.

The DDS control and timing graph describing the actions of the interface upon receipt of a read or write signal is shown in Figure 5.20. Processing begins at the first p point at the top left. The first action is the initialization of *xack.set* to zero, shown by the binding *xack.set*$\leftarrow 0$. Next is a begin-loop point b corresponding to the outer processing loop. The following interval represents a *wait* for the assertion of one of two mutually exclusive asynchronous inputs *read* and *write*. The inputs are represented by asynchronous predicates bound to the interval. If *write* is asserted a branch (indicated by the dashed line) is made to *or_j1*. After the assertion of *write*, the value *dati.l* is placed on either the *control* or *data* line, depending on the value of the input *adr1.l*. The conditional branch then converges, and the next interval specifies an output of 1 on the line *xack.set*. *Xack.set* must be set to 0 in the next interval before waiting for the asynchronous input *mwtc.l* to take on the value 0. When this value is detected, the outer processing loop is repeated, represented by a return, via point *or_j5*, to the end-loop point e. The behavior that occurs if the asynchronous predicate *read* is asserted instead of *write* is very similar.

An algorithmic state diagram is given instead of the corresponding state table produced from the DDS specification (Figure 5.21). For readability, control outputs are specified in the figure by the data transfer being controlled. The waits on asynchronous inputs are implemented as loops at *s2*, *s10*, and *s6*. The conditional behavior based on the value of input *adr1.l* is implemented as conditional outputs during states *s7* and *s3*. Ten states are required, including loops.

For the same example, ISYN produces an event schedule with 26 steps, including some "waits", or implicit loops. Many of the steps serve merely to transfer control to the next step. The elimination of these empty steps would be complicated because of the need to double check timing constraints. The maximum length path (excluding loops, present in both) for the control is six states for our

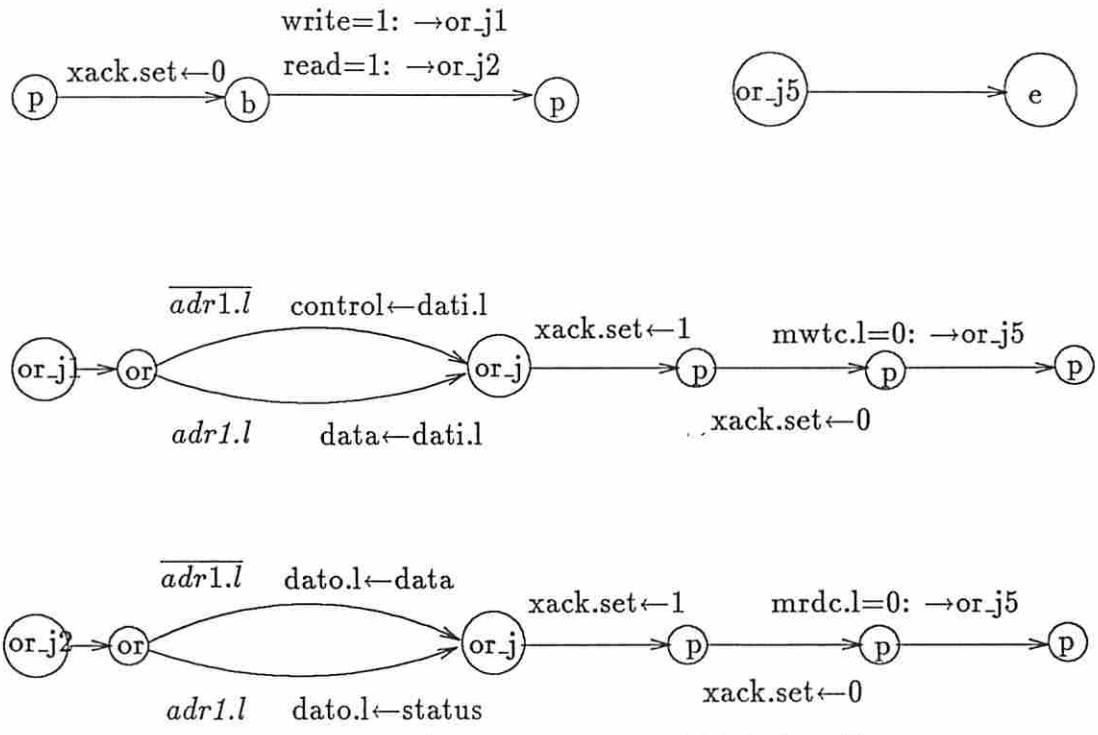


Figure 5.20: DDS Representation of Multibus Slave

design versus eleven in ISYN's. This example state machine was generated in 3 seconds wall clock time on a Sun 3/280.

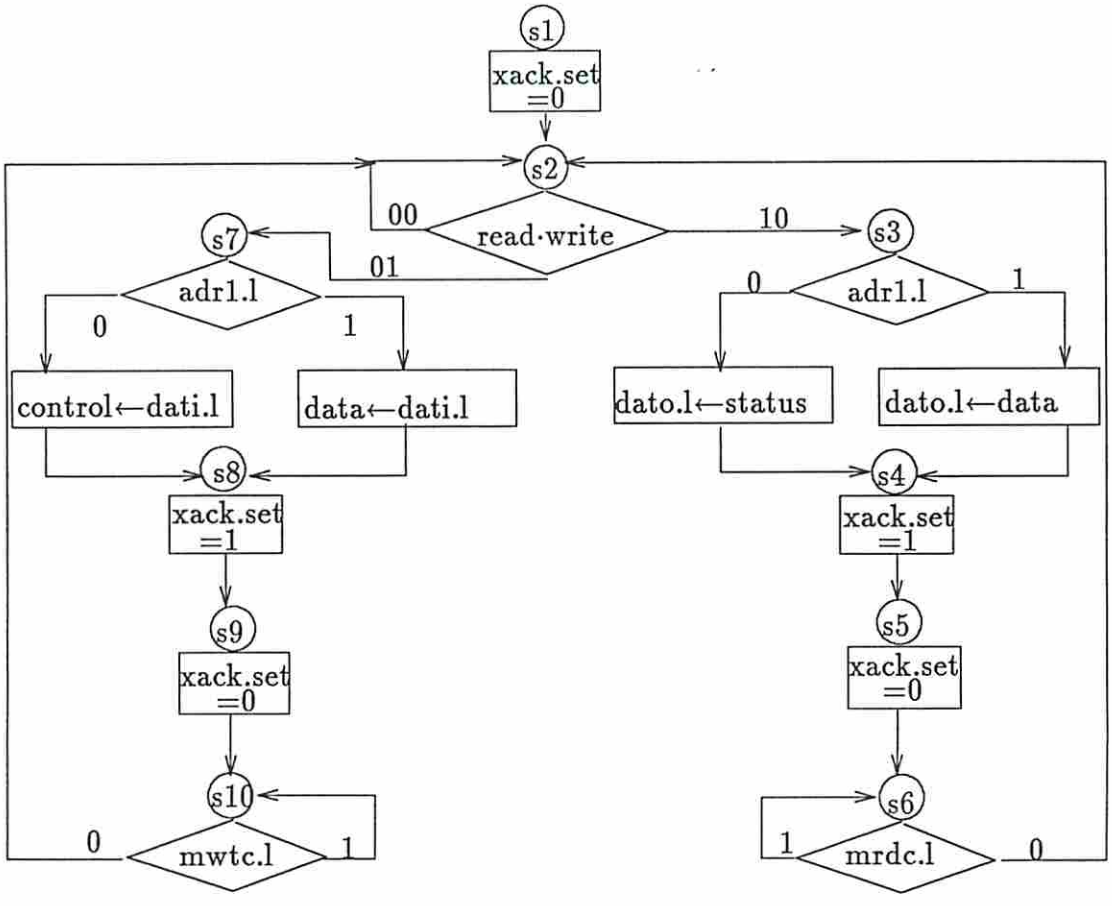


Figure 5.21: State Diagram Derived for Multibus Slave Example

Chapter 6

Conclusions

This research has included the development of methods for the representation and synthesis of digital specifications with complex control and timing along with data manipulation. A study was made of the types of behavior typically implemented using ASIC technology, from which a behavioral schema was established to guide the design system development. Existing methods and conventions were extended to represent the behavior, and a new synthesis technique was developed. The synthesis approach was validated by a software implementation called CONSPEC, and by comparing the automatic designs to manual designs and to the output of other design automation programs.

6.1 Contributions

The representation work established solutions to several previously unresolved representation problems for high-level specifications with significant interface functions and complex control in addition to data manipulation. In addition, CONSPEC's representation and synthesis methods are useful and novel because the behavior that is synthesized incorporates control of the data path as well as behavior that is not related directly to the data path, including communication protocols and control functions related to the external environment. Both asynchronous and synchronous signals can be present in the same specification. One conclusion of this research is that the algorithmic state machine is a good model for the synthesis

of interfacing processors. Internal representations like the DDS that have separate representations for data flow and timing and control behavior are therefore especially suited for the synthesis of interfacing processors.

CONSPEC's method for the satisfaction of minimum time constraints is superior to an operation scheduling approach because states are added only when required to meet the constraint without lengthening paths more than necessary and because timing states are shared between conditional branches whenever possible. Shorter paths through conditional branches result in better average execution speeds, while the sharing of timing states reduces the need for costly optimization procedures following state table derivation.

Maximum time constraint satisfaction is performed using design techniques that had not been previously automated, namely the early assertion of conditional outputs and clock suppression. The user may also interactively reduce the clock period to attempt to meet maximum time constraints if these methods fail. In addition, subroutine and procedure calls are implemented without extra control steps, thus resulting in shorter overall execution times. Loops are considered in calculating path length for time constraint satisfaction.

CONSPEC implements parallel control branches by choosing one of two possible methods, depending on problem characteristics. Either a separate FSM is created to implement each branch, or a single FSM is established to execute the actions of all branches. The use of a single machine can reduce the number of states as well as eliminate handshake wires for simple parallel branches.

The design system has knowledge of several potential interactions between individual design decisions and is capable of resolving most of them. For those cases in which negative interactions lead to failure, CONSPEC has some elementary error analysis capability, alerting the user to the existence of the interaction. Explicit goal based reasoning increases the flexibility of the design approach: the methods developed for goal and interaction representation, detection, and solution can be applied not only to the goals and phases supported by the current version of CONSPEC but to others as well. New goals and/or plans may be added and other design phases supported, thereby expanding the range of explicit goal-based reasoning and increasing the capabilities of the design system in a modular fashion.

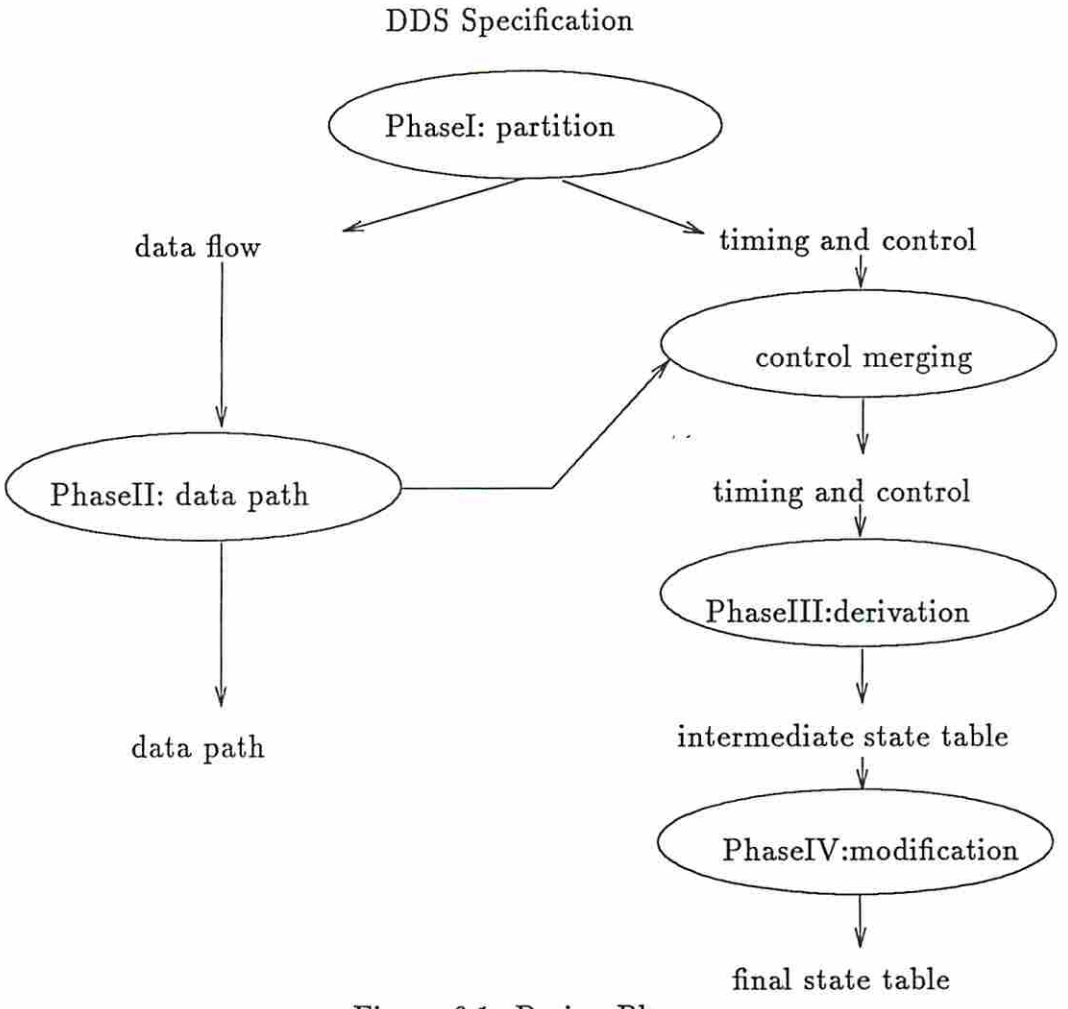


Figure 6.1: Design Phases

Finally, the specification of the control using finite state machine techniques supports well-known optimization methods capable of producing reliable designs and permitting multiple implementation styles.

6.2 Future Research

Figure 6.1 shows the four design phases partition, data path, (design) derivation, and modification. Although only the derivation and modification phases are supported by this version of CONSPEC, which focuses on control design, the explicit goal-based reasoning strategies described in this chapter could be usefully applied to other design phases.

The partition of behavior into data flow and control is not absolute and determinate, but relative and variable. For a single specification, there is a range of problem decompositions and corresponding implementations: from small controllers with a large data path, to a relatively large controller and smaller data path.

> The definition of what is architecture and what is algorithm is arbitrary. In general, by enlarging the architecture, we may convert an ASM into one with fewer but more complex states. At the limit, it is always possible to describe any algorithm in a single state. [WP80] (p.186)

The best problem decomposition is defined by the type of behavior as well as the desired performance/cost characteristics of the implementation. Chapter 3 described canonical methods of translating from SLIDE constructs to the DDS. A DDS description that is derived from a SLIDE specification using these methods defines one possible way to partition the specified behavior into separate data flow and control graphs. Data flow graph behavior is implemented using data path synthesis techniques, while control and timing graph behavior is implemented by CONSPEC using finite state machine techniques. For this reason, the form of the DDS representation does have an impact on the type of implementation derived.

A canonical SLIDE to DDS translation will create an acceptable partition for many specifications. However, for a particular specification it may be desirable to explore alternative data flow/control partitions through transformations on the data flow and timing graphs. The motivation for this exploration may be violated design constraints such as time and area. The reduction of data flow bit width, for example, is one way to reduce area. The cost of such a modification is a more complex control and increased processing time. Another motivation to explore alternate data flow/control partitions is the possibility for optimization, as can be seen in [Jai89] and [Mli]. For example, the values of a sequence of input bits can be tested using a data path shift register and comparison gates or by the controller through a sequence of conditional state transitions. One or other of these implementations may be clearly superior, depending on the size of the controller and number of bits to be tested.

Explicit goal based reasoning could be usefully applied during the partition phase to detect behaviors such as these that can be variously partitioned between

data flow and control. Alternate plans for each goal would define different partitions.

The data path synthesis phase involves module selection, scheduling, and module allocation and binding. It is currently assumed that this is performed using some existing design system such as MAHA, although the interaction with data path synthesizers needs to be automated. If a number of data path synthesis systems or techniques with different characteristics were available, it would be possible to represent data path synthesis steps as explicit system goals with the different solution techniques as plans. The data path synthesizers should be able to design with sequential components such as counters and shift registers, an important capability for ASIC design that is not supported by any system we are familiar with.

During the modification phase interactions are handled between two goals at a time. A single plan can be constrained by multiple order execution meta-plans, but by only one "restrict range" meta-plan. If one plan overlaps with two others, for example, it is not possible to constrain the plan to avoid both overlapping ranges. The restrict range meta-plan should be modified to consider interactions between multiple goals, possibly by accumulating the ranges to be avoided in a list for each plan.

Several new goals or plans could be defined for the modification phase. For example, it is sometimes possible to use a data path value directly as a control signal for a data path event. This simplifies the design because the data path value is no longer input to the controller, and the controller asserts one less conditional output. As another example, a counter or delay element may be used in the data path in place of a state chain for timing purposes. The output of the counter is compared to some limit, and the output of that operation is input to the controller. The controller is able to use a single state loop in place of the state chain. Finally, it is sometimes possible and desirable to unwind control loops. CONSPEC could be expanded to handle these situations and others by specifying explicit goals and the associated plans for their realization.

Reference List

[BD86] M. Bushnell and S. Director. VLSI CAD tool integration using the ULYSSES environment. In *Proc. of the 23rd Design Automation Conf.*, pages 55–61, June 1986.

[BK87] G. Borriello and R. Katz. Synthesis and optimization of interface transducer logic. In *Int'l Conf. on VLSI*, pages 274–277, August 1987.

[Cha87] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[Com84] D.J. Comer. *Digital Logic and State Machine Design.* Brigham Young University, 1984.

[Dav87] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, 1987.

[Dec86] R. Dechter. Learning while searching in constraint-satisfaction-problems. In *Proc. of the AAAI*, pages 178–183, August 1986.

[DMK88] G. De Micheli and D. Ku. Hercules–a system for high-level synthesis. In *Proc. of the 25th Design Automation Conf.*, pages 483–488, 1988.

[FL86] F. Fonsalas and J. Lejard. An edge enhancement algorithm using a polynomial transformation. *Trans. on Consumer Electronics*, August 1986.

[Gir84] E. Girczyc. *Automatic Generation of Microsequenced Data Paths to Realize Ada Circuit Descriptions.* PhD thesis, Carleton University, 1984.

[GK84] E. Girczyc and J. Knight. An Ada to standard cell hardware compiler based on graph grammars and scheduling. In *Proc., 1984 Int'l Conf. on Computer Design - ICCD*, pages 726–729, October 1984.

[GKP85] J. Granacki, D. Knapp, and A. Parker. The ADAM advanced design automation system: Overview, planner and natural language interface. In *Proc. of the 22nd Design Automation Conf.*, pages 727–730, June 1985.

[GR87] R. Galivanche and S. Reddy. A parallel PLA minimization program. In *Proc. of the 24th Design Automation Conf.*, pages 600–607, June 1987.

[Gra86] J. J. Granacki. *Understanding Digital Systems Specifications Written in Natural Language.* PhD thesis, University of Southern California, November 1986.

[HPG88] S. Hayati, A. Parker, and J. Granacki. Representation of control and timing behavior with applications to interface synthesis. In *Proc. of the Int'l Conf. on Computer Design*, pages 382–387, 1988.

[Jai89] R. Jain. *High-Level Area-Delay Prediction with Application to Behavioral Synthesis.* PhD thesis, University of Southern California, July 1989.

[JKMP89] R. Jain, K. Kucukcakar, M. Mlinar, and A. Parker. Experience with the ADAM synthesis system. In *Proc. of the 26th Design Automation Conf.*, pages 56–61, June 1989.

[JPP88] R. Jain, A. Parker, and N. Park. Module selection for pipelined synthesis. In *Proc. of the 25th Design Automation Conf.*, pages 542–547, June 1988.

[Keu89] K. Keutzer. Three competing design methodologies for ASIC's. In *Proc. of the 26th Design Automation Conf.*, pages 308–313, June 1989.

[Kow84] T. J. Kowalski. *The VLSI Design Automation Assistant.* PhD thesis, Carnegie-Mellon University, April 1984.

[KP] K. Kucukcakar and A. Parker. Mabal: A software package for module and bus allocation. To appear in the Int'l Journal of Computer-Aided VSLI Design.

[KP83] D. Knapp and A. Parker. A data structure for VLSI synthesis and verification. Technical report, Digital Integrated Systems Center, Dept. of EE-Systems, University of Southern California, October 1983.

[KP85] D. Knapp and A. Parker. A unified represention for design information. In *Proc. of the IFIP Conf. on Hardware Description Languages*, August 1985.

[KT83] T.J. Kowalski and D. Thomas. The VLSI design automation assistant: Prototype system. In *Proc. of the 20th Design Automation Conf.*, pages 479–483, 1983.

[LL84] J. L. Lewandowski and C. L. Liu. A Branch and Bound Algorithm for Optimal PLA Folding. In *Proc. of the 21st Design Automation Conf.*, pages 426–431, June 1984.

[McF78] M. McFarland. The value trace: A data base for automated digital design. Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., December 1978.

[McF81] M. McFarland. Allocating Registers, Processors, and Connections. Internal Paper, CMU, 1981.

[McF83] M. McFarland. Computer-aided partitioning of behavioral hardware. In *Proc. of the 20th Design Automation Conf.*, pages 472–478, June 1983.

[Mli] M. Mlinar. Ph.d. thesis. In progress.

[MP] M. Mlinar and A. Parker. Loop transformation for acyclic data path synthesis. Internal USC Document.

[MP83] M. McFarland and A. Parker. An abstract model of behavior for hardware description. *IEEE Trans. on Computers*, C-32(7):621–637, July 1983.

[MSVV83] G. De Micheli, A. SanG.-Vincentelli, and T. Villa. Computer-aided synthesis of PLA-based finite state machines. In *Proc. of the ICCAD*, pages 154–156, 1983.

[Nes87] J. Nestor. *Specification and Synthesis of Digital Systems with Interfaces.* PhD thesis, Carnegie-Melon University, 1987.

[Nii86] H. P. Nii. The blackboard model of problem solving. *AI Magazine*, pages 38–64, Summer 1986.

[NT86] J. Nestor and D. Thomas. Behavioral Synthesis with Interfaces. In *IEEE Int'l Conf. on CAD*, pages 112–115, November 1986.

[Par75] A.C. Parker. *A Generalized Approach to Digital Interfacing.* PhD thesis, North Carolina State University at Raleigh, 1975.

[Peg88] 1988. Internal Berkeley documentation provided with system.

[PH87] A. Parker and S. Hayati. Automating the VLSI Design Process. *Proc. of the IEEE*, pages 777–785, June 1987.

[PP85] N. Park and A. Parker. Synthesis of optimal clocking schemes. In *Proc. of the 22nd Design Automation Conf.*, pages 489–495, June 1985.

[PP86] N. Park and A.C. Parker. Sehwa: A program for synthesis of pipelines. In *Proc. of the 23rd Design Automation Conf.*, pages 454–460, July 1986.

[PPM86] A. Parker, J. Pizarro, and M. Mlinar. Maha: A program for datapath synthesis. In *Proc. of the 23rd Design Automation Conf.*, pages 461–466, July 1986.

[PW81] A. Parker and J. Wallace. SLIDE: An I/O Hardware Descriptive Language. *IEEE Trans. on Computers*, C.30(6), June 1981.

[Sac77] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, pages 115–135, May 1977.

[SR88] J. Stankovic and K. Ramamritham. *Tutorial: Hard Real-Time Systems.* IEEE Computer Society Press, 1988.

[Sta80] T. A. Standish. *Data Structure Techniques.* Addison-Wesley, 1980.

[Ste81] M. Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16:111–139, May 1981.

[The76] A. Thesen. Heuristic scheduling of activities under resource and precedence restrictions. *Management Science*, pages 412–442, December 1976.

[Ton88] C. Tong. *Knowledge-based circuit design.* PhD thesis, Stanford University, 1988.

[Wil83] R. Wilensky. *Planning and Understanding.* Addison-Wesley, 1983.

[Wil84] D. E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 1984.

[Win84] P. H. Winston. *Artificial Intelligence.* Addison-Wesley, 1984.

[WP80] D. Winkel and F. Prosser. *The Art of Digital Design.* Prentice-Hall, 1980.

Appendix A

Representation of Control and Timing Behavior

A.1 Introduction

Synthesis[1.1] is defined as the automatic design of a register transfer level structure from a description of the desired system behavior. Synthesis systems have traditionally worked from data flow graphs annotated with simple timing specifications. Digital systems with complex timing and control requirements have been the focus of only a few systems which synthesize interface circuits[BK87, NT86]. Because of the complexity involved there are many unresolved issues in the representation of such systems, whose characteristics include both synchronous and asynchronous events, timeouts, delays, and process priorities. One fundamental difficultly is the representation of both data flow and control behavior in a single specification. This paper describes a representation for systems such as interfaces whose specifications include both data flow and control and timing information, using the *Design Data Structure* (DDS)[Gra86, KP85], which was developed at the University of Southern California where it is used by several synthesis programs.

A.2 Related Research

John Nestor at CMU designed a system for the synthesis of interfaces called Behavioral Specification with Interfaces (BSI) [NT86]. This system uses a modified

[1.1] This appendix is the text of a paper published in ICCD in 1988 [HPG88].

version of the value trace (VT) [McF78], which is a data flow graph that incorporates control information. Nestor added I/O operations plus timing constraints to the VT. Structural information about the ports, including bit width and electrical characteristics, is contained in a separate file. Two port operations and one control operation are represented as operations in the data flow graph. Timing of the port operations is fixed in relation to the clock cycle. The control operation was introduced to specify absolute sequencing information. This was necessary since data dependency alone could determine the sequencing of operations in the VT. Because interface events often have sequencing requirements that are free of data dependencies, representing such events as data path operations requires the graph to be broken at certain points, which is the function the control operation performs. Each BSI timing constraint takes the form of a maximum or minimum time relation between two data path operations, which are referred to by their labels. The specification includes a file listing all the timing constraints associated with the VT.

Borriello at UCB also produced an interface synthesis system called *Janus* [BK87]. Janus accepts two interface specifications from which it produces the interconnecting transducer circuitry. The internal data structure of this system is termed an *event graph* [BK87], which is a timing graph derived from a conventional timing diagram. This is a different approach from the annotated data flow graph method described above. Event graph nodes correspond to transitions in signal logic levels, with two node types indicating input and output. Simultaneous events are combined in a single node. There are two arc types: the first indicates sequencing constraints, the second signifies timing constraints between two events. Conditionals are represented implicitly by the connections between events. Signals may be described by Boolean expressions if that is more natural than the timing diagram approach. Synchrony is specified by identifying each node as either asynchronous or synchronous, with a clock specification in the latter case.

A.3 The Design Data Structure

The Design Data Structure represents specifications and implementations using separate *subspaces* for the dataflow behavior, timing and control behavior, logical

structure, and physical structure. Each subspace may be hierarchically decomposed, but the decomposition may not be isomorphic. This is illustrated in Figure A.1, where Fetch and Execute can both be decomposed into more detailed behavior. The CPU and memory can be decomposed likewise. However, both Fetch and Execute make use of CPU *and* memory. Items in one subspace may be related to those of other subspaces by bindings, for example, between the addition in the dataflow, the ALU in the structural subspace which is used to implement it, and the time range during which the addition is performed. All relationships between the subspaces are explicitly delineated in this way.

The representation of a wide range of specifications which fuse varying amounts of data flow and control is possible. The data flow and the timing and control information are described separately, but the two are integrated in a well defined way. This method has the advantage of clearly differentiating between the two design elements while allowing a unified synthesis approach. This is desirable because the design of data path and control is commonly done separately to reduce complexity, and yet tradeoffs between the two are possible and often performed. Furthermore, each subspace has a degree of independence from other subspaces. A single dataflow graph has many possible timing behaviors with various degrees of parallelism, and may be implemented by different module sets. The use of different subspaces therefore supports the creation of multiple designs from the same specification.

In DDS, abstract system behavior is represented using the data flow and timing subspaces. Synthesis systems manipulate this information and also place additional implementation information in the structural and physical subspaces to create the design. This paper shows a way in which the DDS timing subspace can be used in conjunction with the data flow and structural subspaces to represent interface behavior. Much interfacing behavior can be represented as the relative timing of signal events, as the success of Borriello's method demonstrates. Representing such events as operations in a data flow graph as Nestor has done is more unwieldy than the timing graph approach. Nevertheless, interface behavior may also include operations of a data flow nature such as those seen in network protocols, in which case timing graphs prove inadequate. The advantage of DDS is that both data flow and timing graph representations are employed in a unified way. We will first give

Figure A.1: The Four Subspaces

a brief description of the DDS data flow and timing subspaces and then present some categories of interface behavior with the corresponding DDS representations. A small interface specification and its DDS form will then be given.

The *data flow subspace* (DFss) represents data operations, their inputs, and the data dependencies between operations. Two node types, operation and value, are connected by directed arcs to form a bipartite directed acyclic graph. This is a single-assignment representation, so the names given to values do not indicate variables and cannot take on different values at different times. To account for cycles, a subscript may be affixed to value and operation names, thus ensuring a unique name for each value. The lifetime of a value can be represented by a *binding* from the value node to a time interval in the timing subspace. To represent a variable, a sequence of values which exist at different time intervals are all bound to a single line or register represented in the structural subspace. Constants, such as the digital values of zero and one, may be represented by value nodes in the data flow subspace. All subspaces in the DDS can be *hierarchically composed.* In the DFss, abstract operation nodes must vultimately be decomposable into library primitives such as Boolean, relational, and simple arithmetic operations. Primitives are described by truth tables in the system library.

The *Timing and Sequencing Subspace* (TSss) is a directed acyclic graph whose points represent events, such as the initiation or termination of an operation, and whose arcs represent relations between events, such as operation durations and causality. Points cannot be hierarchically composed, but arcs may represent complex processes which are decomposable into subarcs and associated points. There are four types of arcs and seven point types in the model. Arcs and points are given names which indicate their type, affixed with a number to differentiate multiple objects of the same type, if necessary. Symbols used in this naming convention are given in parentheses for each object. An *interval arc (i)* is used to specify a range of time. When bound to an operation in the DFss, it represents the time interval during which that operation occurs. A specific time value may or may not be assigned to the arc; in any case an interval arc denotes the relative occurrence of the operation. Other arc types include the *time constraint arc (tc)*, the *causal arc (ca)*, and the *inertial delay arc (d)* which is used to represent the delays of physical components. The significance of these arcs will be described below.

The simplest points (*p* or unlabeled) join two arcs, one entering and one leaving. A *cobegin point (and)* has one arc entering and more than one exiting; a *coend point (and_j)* has multiple arcs entering and one leaving. The semantics of these points signifies that co-processes begin and/or end simultaneously. The two points are not required to occur in pairs. An *or-fork point (or)* is an exclusive-or and is always associated with a *join point (or_j)* signifying the end of the branch. Each arc leaving an or point is associated with a *synchronous predicate* that determines which arc will be traversed: the predicate of only one arc can be true at a time. Loops are represented using two point types: a *begin-loop (b)* and an *end-loop (e)*, between which are located all the events and intervals of the loop. All points and arcs within the range of the loop are subscripted to differentiate between iterations, as this is a single assignment representation.

A.4 Interface Behavior

Interface behavior involves the reading and writing of asynchronous and synchronous signals which possess complex timing and control flow. However, it also often involves data manipulation such as address comparison and data encoding.

Here we describe different aspects of interface behavior and methods of representation in DDS.

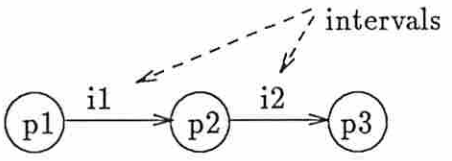
Timing information includes the proper sequencing of and durations between events. In the DDS, operation precedence based on data dependency is depicted in the data flow subspace, which reveals the maximum parallelism. This data dependent sequencing is not reflected in the timing subspace unless the scheduling process assigns operations to different steps. Portions of a data flow graph where there are no conditional branches or interior loops and no timing constraints between operations within the subgraph are initially bound to a single interval in the timing subspace. Events which are scheduled sequentially are bound to adjacent time ranges; simultaneous events are bound to a single time range; events which occur in parallel are bound to arcs on different branches of a *cobegin* point (Figure A.2). There is no explicit time relationship between different points on parallel branches, except where the branches come together at a *coend* point.

To represent more general time constraints between events, such as "Operation X must occur within z seconds after operation Y," a *time constraint* arc is used. This arc specifies only timing information and is not bound to any operation; the information may take the form of an equality or an inequality specifying a minimum and/or maximum time. In addition, an *interval arc* can be assigned a value for a specific time range, A *causal arc* directed from event A to event B indicates that the occurrence of A (such as the termination of an operation) caused or initiated event B. There is no time associated with a causal arc, and operations cannot be bound to it.

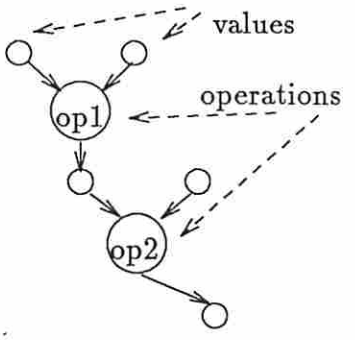
A.4.1 While Statement

More complex control flow such as "while" and "if" constructs are also represented largely in the TSss. Figure A.3 shows the DDS template for the statement: *While "cond" do "op"*. The causal arc is required in the TSss because the semantics of the begin loop point do not allow multiple outgoing arcs. The operation "op" is represented in the DFss, along with its input and output values. The index j defining the loop in the TSss is also attached to the "op" node, as well as to all its inputs and outputs. Outputs with incremented indices indicate that these outputs

Timing and Sequencing Subspace

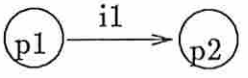


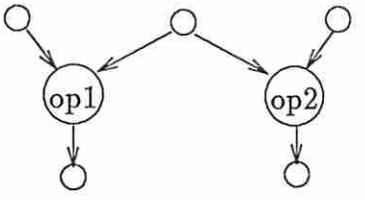
Data Flow Subspace



a. Sequential Events

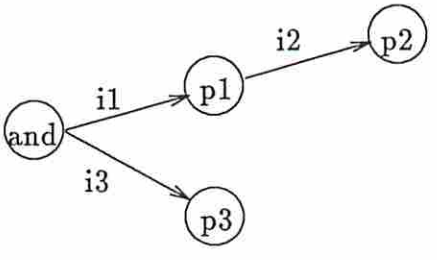
Binding(op2, i2)
Binding(op1, i1)

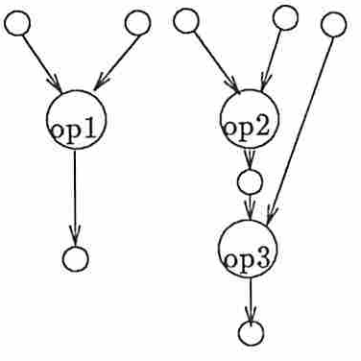




b. Simultaneous Events

Binding(op1, i1)
Binding(op2, i1)





c. Concurrent Events

Binding(op1, i3)
Binding(op2, i1)
Binding(op3, i2)

Figure A.2: Event Ordering

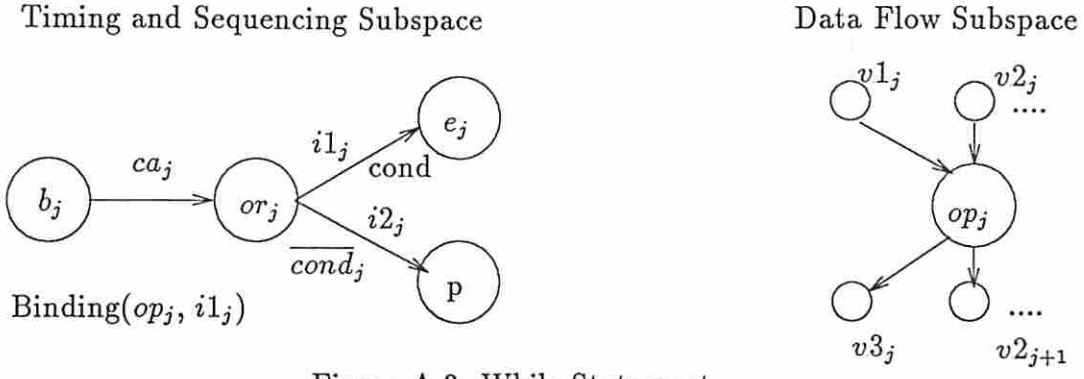


Figure A.3: While Statement

are fed back into the next loop. The condition "cond" serves as the predicate on the or-fork point's two arcs. If the predicate is true, arc $i1_j$ is traversed. Because "op" is bound to this arc, it is executed. The loop index j is then incremented and the loop is reentered. If the predicate is not true, arc $i2_j$ is traversed and the loop is exited. Note that because the operation is bound to an arc following the or-fork point, if the predicate is initially false, the operation is never performed. If instead "op" is bound to an interval preceding the or-fork point, the effect would be that of a *Repeat "op" until "cond"* statement, in which the operation would be performed at least once. Both of these examples illustrate the representation of the general conditional statement *If "cond" then "op1" else"op2"*, which uses the or-fork point. The conditional value, which may be computed in the data flow subspace, acts as a predicate which chooses between two intervals: one which is bound to op1 and the other to op2.

A.4.2 Detailed Timing

The DDS can represent detailed timing behavior down to circuit level details such as setup and hold times. The highest level representation of an operation or event would be a single node in the DFss which is bound to a single time interval of unspecified length in the TSss. This can be refined to the desired extent by decomposing the top level arc into intervals bound to constituent events and time constraints between them, such as setup time. An inertial delay arc can be used in place of an interval arc to clearly indicate the nature of such a time interval.

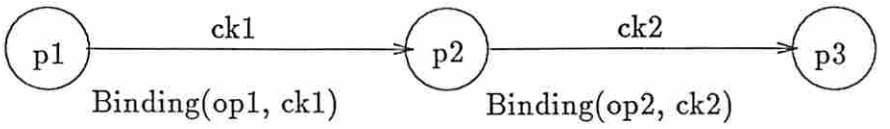


Figure A.4: Synchronous Behavior with Clocking

A.4.3 Clocking

Interface behavior often involves *periodic signals* and the relationship of signal changes to clock levels is often a crucial part of that behavior. In the DDS, clock periods are considered to be a special form of time interval with arcs designated by the symbol *ck* and associated with a "length" specifying a time period, if known. The TSss of a synchronous design would therefore include a sequence of clock period intervals, to which operations may be bound (Figure A.4). For those applications in which clock levels are related to system events, such as inputs which change on a rising edge and conditional outputs asserted on the falling edge, a distinction is made between the low and high clock phases, represented by arcs labelled *lo* and *hi*, respectively.

A.4.4 Asynchronous Signals

Asynchronous signals may occur in hardware, particularly in interfaces. An example of such a signal is reset, which can invoke certain actions, regardless of the state of a system. These are represented in DDS by a Boolean expression which determines conditions under which the asynchronous event will occur, and a destination point to which processing will branch in that event. In particular, a special binding is defined between three subspaces, depicted by {Binding(carrier, value, interval), point} (Figure A.5). "Carrier" is a structural element or line, "value" is defined in the data flow subspace, "interval" is a time interval arc in the TSss, and "point" is an exclusive-or join in the TSss. This binding signifies that if a certain value is asserted on the asynchronous predicate's carrier at any time during the specified time interval, a branch occurs to "point" which defines the start of subsequent processing.[1.2] The interval which the asynchronous predicate is bound to

[1.2]An exclusive-or join is used for "point" because the underlying semantic model assumes an infinite number of conditional branches along the interval over which the predicate is defined [Gra86].

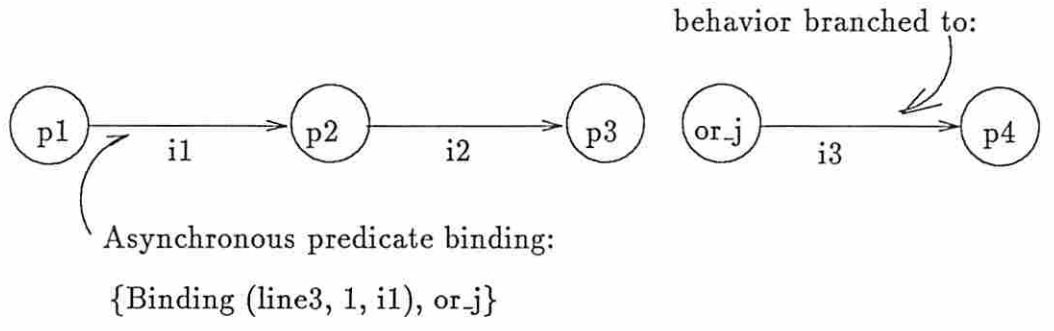


Figure A.5: Asynchronous Behavior

may itself be decomposed into smaller intervals, any of which may be exited from if the asynchronous predicate becomes true. In figure A.5, *i1* is the interval to which the binding is attached, and *or_j* the point to which branching occurs. Interval *i3* has bound to it those operations performed if the asynchronous signal is asserted. Conversely, if interval *i1* terminates before the asynchronous signal occurs, processing continues with the operations bound to interval *i2*. Many asynchronous predicates may be bound to the same interval, provided that they are mutually exclusive. Asynchronous signals may occur in a synchronous specification and are easily represented in conjunction with synchronous events using the DDS. In that case the signal's asynchrony is defined in terms of a system clock. In the absence of a clock, asynchronous behavior occurs when signal timing is indeterminate relative to other system events.

A.4.5 Process Priority

Process priority, in which higher priority processes can preempt and terminate lower priority ones, may be represented using the asynchronous predicate construct as shown in Figure A.6. The interval arc to which the "preempt" signal is bound is hierarchically composed of the lower priority process's procedures, all of which inherit the asynchronous predicate. If the predicate becomes true, the low priority process terminates and a branch occurs. This is a representation of the abstract behavior. The actual mechanisms depend on the implementation chosen.

Process initiation is represented in a similar manner. The initiating condition or predicate is bound to a time interval. If the sensing and processing of this predicate proceeds in parallel with other events, all events are tied together with and points,

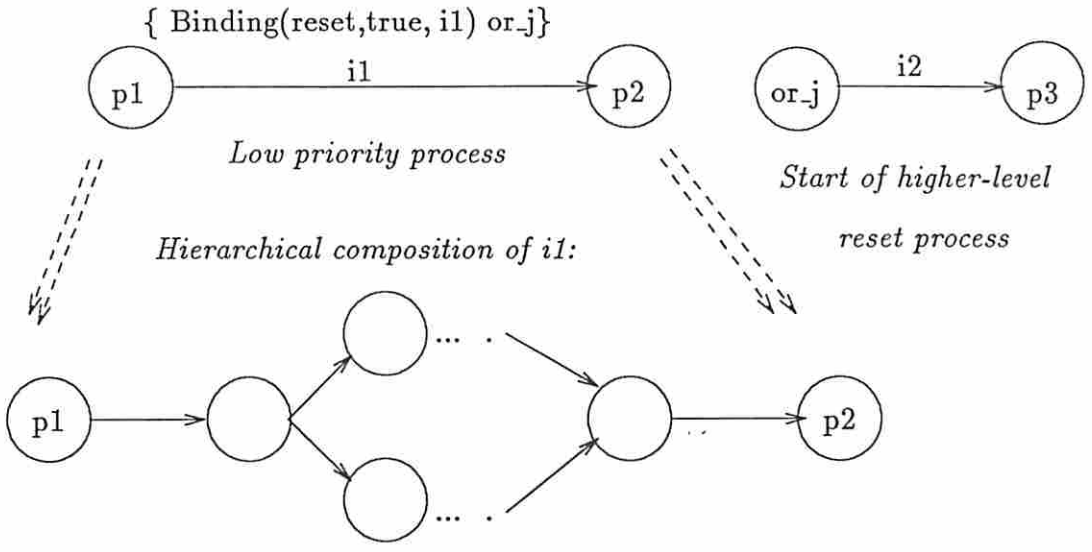


Figure A.6: Process Priority

as shown in Figure A.7. If the predicate is asserted, the behavior bound to that arc terminates and a branch is made to the first point of the initiated process. This does not terminate the processes bound to other arcs of the higher level process, however.

A.4.6 Delays and Timeouts

Delays and *timeouts* can be concisely represented in the DDS. The simplest delay is a pause in processing for a specific time period, which could be expressed in a hardware descriptive language as: *Delay "time"*. In DDS this is represented using a time constraint arc, as shown in figure A.8. A more complex delay is a wait on a predicate: *Delay until "predicate" equals {0,1}*. This is represented using the asynchronous predicate construct illustrated in figure A.5 except that *i1* has no time limit and *i2* is absent, since processing is suspended until the signal is asserted.

Using this form of the delay statement, if the proper signal value is never asserted the system waits forever. To avoid this, the specification may include a time-out clause:

Delay "time" until "predicate" equals {0,1}{then...}{else...}

Here the optional then clause specifies a response to the assertion of "predicate" and the else clause, also optional, indicates actions to be performed in case of

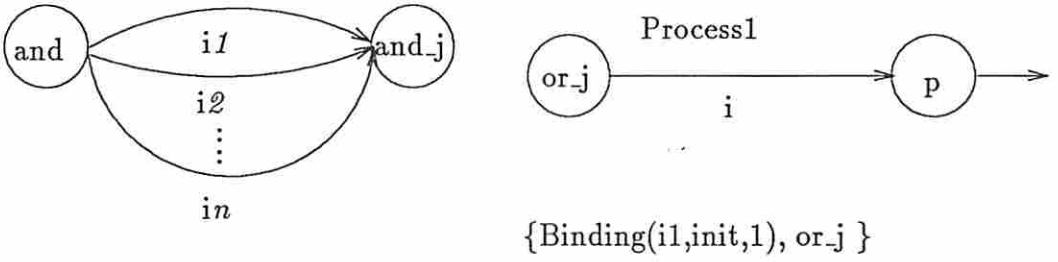


Figure A.7: Process Initiation

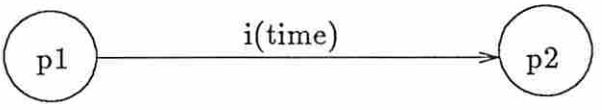


Figure A.8: Delays

a timeout. The corresponding DDS representation is shown in figure A.9. The first interval of the delay statement, *i1*, has been assigned a time length. The asynchronous predicate is bound to this limited interval. If the time elapses without assertion of the proper signal a time-out occurs, in which case the system proceeds to the else clause operations bound to *i2*. If no else clause is present, *p2* and *i2* are omitted. Conversely, if the asynchronous predicate *is* asserted some time during *i1*, processing branches to *or_j1* and the operations bound to *i3* are executed. If no then clause is present *i3* and *or_j1* are omitted and the system branches to *or_j2* when the asynchronous predicate is asserted. The last interval in this figure, *i4*, is bound to whatever operation follows the delay statement. If either of the else or then clauses is a branch statement the corresponding arc will be directed elsewhere in the timing graph.

A.4.7 Physical Lines and Registers

Interface behavior descriptions, such as asynchronous predicates, include references to *physical lines* and *registers*. The setting of lines to specified levels is represented

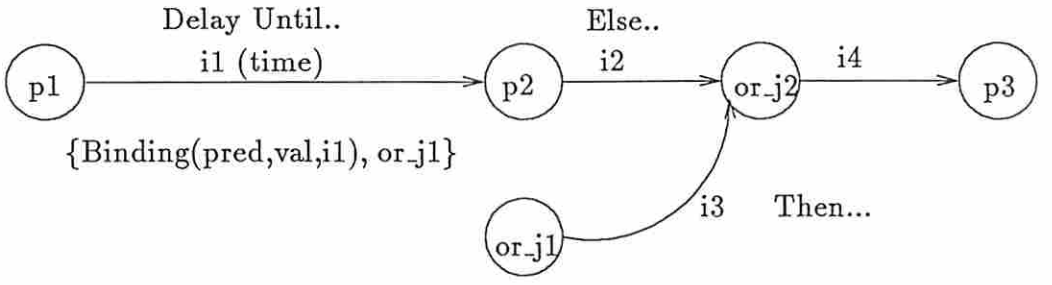


Figure A.9: Time-outs

in DDS using bindings between the signal carrier, a value defined in the data flow subspace, and a time interval, designated by Binding(carrier,value,interval) . This binding indicates that the specified value must be asserted on that carrier. The value is asserted from the start of the specified time interval until a change is indicated by a binding defined on a succeeding interval. Tri-state lines require in addition a value with a special system definition to indicate when the line is not to be driven.

The reading of lines or registers which are assigned values from outside the system being described involves similar bindings but requires a different approach. Values read from global lines are defined as variables in system specifications. Because the DDS uses a single assignment data flow structure, it is necessary to associate values with loop indices to convey the same information. Global values are therefore defined in the data flow subspace as indexed values bound to a structural carrier. The time interval during which each indexed value is alive is defined by loop constructs in the timing subspace. If the signal is periodic, its index is updated every clock cycle. Global values can be used as operation inputs or they can be latched into local registers by specifying a binding between the value and the register during the proper time interval.

A.5 Frame Counter Example

To illustrate some of the concepts discussed above, a small interface specification will be given. Although a design automation system would typically employ a hardware descriptive language, the description will be given in English with the corresponding DDS representation:

> A data acquisition system sends serial data at a 1 kHz bit rate to a recording system. The data is in the form of 4-bit words. Several records, each consisting of a variable number of words, will be transmitted each time the data acquisition unit becomes active. Each record is separated from the following record by the frame separator word 1101 which will never be contained in a data record. A count line from the acquisition unit to the recording system will be asserted to signal that counting of the records should begin and will be deasserted to signal the end of record transmission. A clock signal from the data acquisition unit is available, and transmitted data changes on the negative transition of this clock. The count signal will always be asserted as the first bit of the first record occurs. The maximum number of records to be counted is 200.[Com84]

Figure A.10 shows the TSss and DFss with associated bindings for this specification. To simplify the example, an outer loop to reinitialize "count" has been eliminated. C is the count line and D is the data line. Arcs labelled with the symbols *hi* and *lo*, symbolizing high and low clock phases, are used to indicate that values of C and D are not asserted until the falling edge of the clock. At the first *or* point C is tested: if it is asserted the body of the loop is entered. Otherwise, the body is skipped and the loop is restarted at the next clock period to again check the value of C. At each subsequent *or* point in the graph a check is made on the value present on line D during the low phase of the clock. Whenever a particular value of D indicates that the current word is not the frame separator, the cycle of four clock periods is nevertheless completed before the loop is restarted. The branch followed in that case employs *ck* arcs, which indicate full clock periods and make no distinction between the low and high clock phases, since no tests of input values need to be made. Causal arcs are used here to join points where there is no specific interval of time represented by the arc between the two points.

When the frame separator word is detected, an *and* point is used to indicate that two parallel actions are to take place. The arc $lo7_j$ completes the current cycle and is directed to the last point of the loop, indicating that the next cycle is to begin. The second arc i_j is bound to the increment operation in the DFss which updates "count". Because "INC" must terminate before the next record is

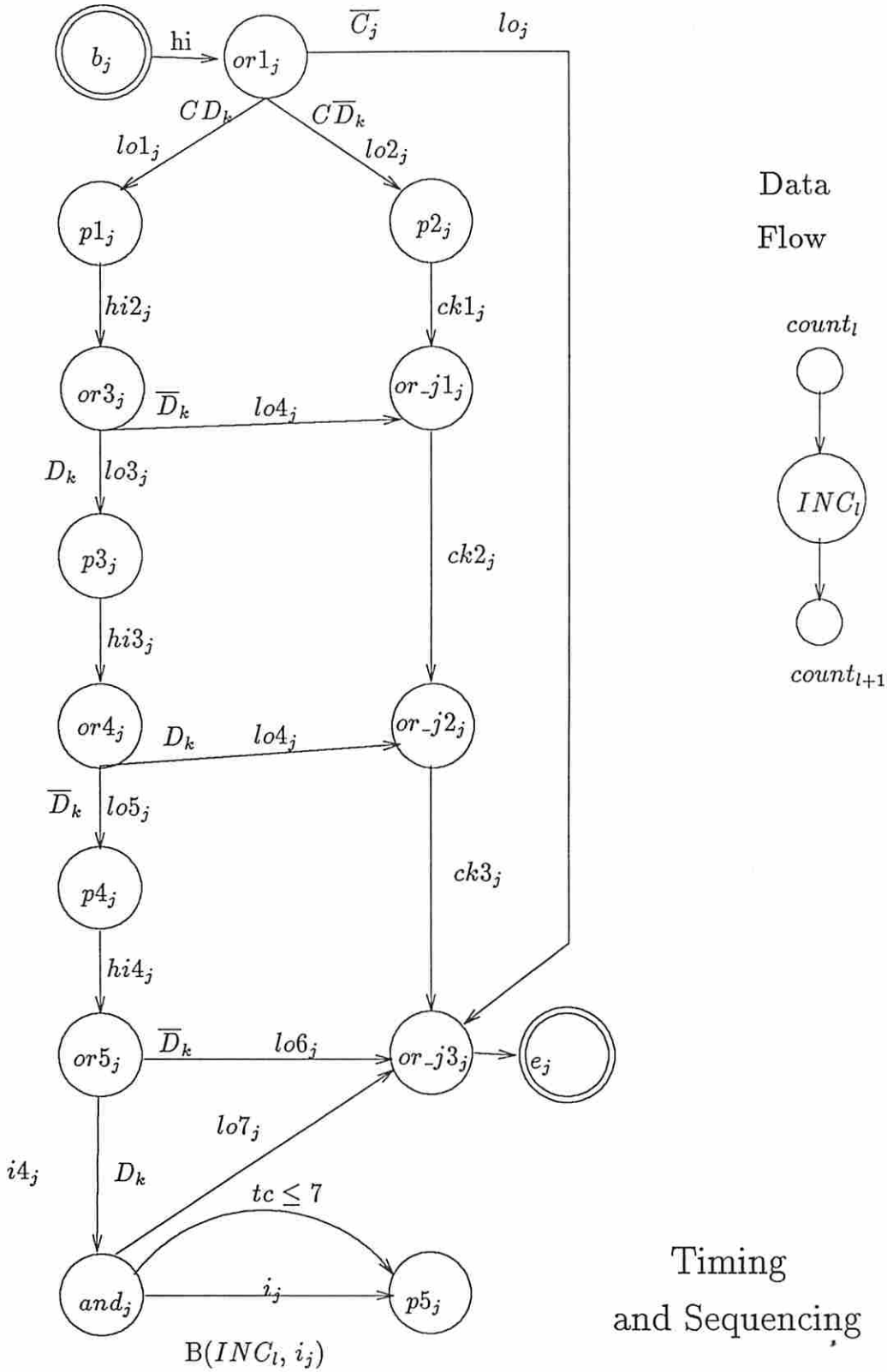


Figure A.10: Frame Counter Example

detected, a time constraint is attached to its interval. On the assumption that each record consists of a least one word in addition to the frame separator, the constraint is set at seven clock cycles. Because the clock rate is specified, each type of clock arc will have a length associated with it. Also, since the maximum size of the value "count" is given, it can be bound to a 16-bit register in the structural subspace.

A.6 Results and Conclusion

This paper has described a formalism for the representation of interface behavior which can be used for synthesis by a design automation system. The Design Data Structure can represent the many facets of interface behavior in a unified way, including timing constraints, synchronous and asynchronous signals, control flow, and data manipulation. Its descriptive power is more complete than some other formalisms in use, including event and annotated data flow graphs. The same representations described here can be used by data path synthesizers to capture more complex timing information than is typically handled and to separate control from data manipulation information to produce cleaner data flow graphs.

The Design Data Structure has been used successfully as an internal representation for a natural language interface for system specification. This application required representation of complex control and timing. The data flow aspects of the DDS have been used successfully by several synthesis systems.