Automatic Production of Controller Specifications From Control and Timing Behavioral Descriptions*

Sally Hayati and Alice Parker
University of Southern California
Los Angeles, CA 90089-0781

Abstract

This paper presents a method for the generation of controller specifications from high-level behavioral descriptions in control and timing graph form. Input descriptions may contain multiple timing constraints, asynchronous and synchronous inputs, data dependent internal loops, and parallel and conditional branches. The timing graph model is transformed automatically to a state table specification of a synchronous finite state machine. The specification method is effective not only for independent data processors, but also for processors constrained by interface requirements and performing I/O protocol translation. The method has been programmed and tested on selected examples. Results from one example are given along with a comparison with results on the same example from another system.

1 Introduction

The design system described in this paper generates controller specifications from a description of data path control signals, I/O signals, branching behavior, and the required timing of events. We are particularly interested in controllers for processors which interact with their environment in nontrivial ways, in addition to carrying out internal processing (referred to here as *interfacing processors*), such as protocol transducers and graphics interfaces. System correctness depends not only on logical results but on their timing as well; time constraints may be multiple and both minimum and maximum. It is also significant that some processor input values may be manipulated by data operations such as multiply or shift to produce new values, and other inputs may be viewed more logically as control signals rather than operator inputs. This dichotomy between control and data is particularly relevant to interfacing processors because of the use of protocols and signalling conventions.

*Supported by the Department of Advanced Research and Project Agency Contract N000 14-87-K-0861

Some design systems approach interface processor synthesis using either data path or control synthesis techniques. ELF [1] was the first data path synthesis system that accepted multiple timing constraints. ISYN [2] schedules data flow and read and write operations based on data readiness and timing constraints without considering hardware sharing. I/O hardware templates are bound to each line read from or written to. The controller is not synthesized. Another system, called Janus [3], designs interface transducers using a timing graph as input. No data path design is performed. The output of Janus is an asynchronous circuit in which timing is determined by the rippling of signals from input to output, using delay elements in the signal path if necessary. A template matching strategy is used to assign hardware to behavior, with little optimization of circuitry.

The control specifier described in this paper is part of the Advanced Design AutoMation (ADAM) system at the University of Southern California. ADAM accepts a behavioral description of a digital processor and generates a register-transfer level design. The control specifier is one element of a subsystem of ADAM under development to synthesize digital interfaces. The *behavioral* input specification of the interface synthesis subsystem may contain a mixture of data manipulation and I/O signal events with timing constraints. The ADAM internal representation, called the DDS, is described in the next section. The DDS maintains separate representations of data flow and timing and control behavior. Data path scheduling and module binding are performed before control specification to provide precise information on the timing and values of all data path control signals. This information will be incorporated into the timing and control behavioral description, which is the input to the control specifier, CONSPEC, whose functioning is described in detail in Section 3. CONSPEC's approach is novel and general in that a state table is generated from which virtually any style of controller can be synthesized. Internal loops are allowed and considered for time constraint satisfaction. Execution time is reduced to the minimum, subject to data path scheduling decisions and minimum time constraints; in particular, there is no time penalty for the use of procedure calls.

The method described here has been programmed and tested on selected examples. Results from one example are given in Section 4, along with a comparison with results on the same example from ISYN. The last section gives conclusions and plans for future research.

2 Representation of Behavior

Within the ADAM system design specifications are represented internally by a graphical data structure called the *Design Data Structure (DDS)*. The significance of this data structure for interfacing processor design is its power to represent the diverse kinds of information needed and the efficiency of its graphical structure at integrating the information into a useful form. The interested reader is referred to [4] for details on representation issues relevant to interfacing processor synthesis.

The DDS represents behavioral specifications using separate hierarchical models for the data flow behavior, timing and control behavior, and logical structure. The representation of a wide range of specifications which contain varying amounts of data flow and control flow is possible. The *Data Flow Model* is a traditional single assignment directed acyclic graph showing maximum parallelism. The two basic node types used are operation and value, within which subtypes are defined. The model of greatest importance to the control specifier is the *Control and Timing (CT) Model*. It is this model which describes the control and timing behavior to be implemented by a controller.

The CT model is a directed 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. There are four arc types that are most relevant for this work. *Time constraint arcs* are used to represent general time constraints between events, such as "Operation X must occur within z seconds after operation Y." The constraint may take the form of an equality or an inequality specifying a minimum and/or maximum time. The *interval arc* is used to specify a range of time: the exact length may be specified or not. *Clock period* and *clock phase arcs* are special cases of interval arcs.

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. The simplest type indicates time order only: events associated with the ingoing arc precede those of the outgoing arc. Parallel branches are specified by *and points* which have a single in-arc and multiple out-arcs that terminate (possibly) at *and-join points*. Conditional branches employ *or points*, the branches of which terminate at *or-join 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. Loops, which may be nested, are represented 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. For brevity, subscripts are omitted within figures and points are labelled with the first letter of the point type, such as *b* for a begin-loop point. Simple points are labelled with a *p* or left unlabelled.

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. In the next section we explain the nature and significance of time interval bindings along with details of the control specifier.

3 Control Specification

We now describe the control specification method used by CONSPEC. A major goal is to retain flexibility in choosing the controller implementation style rather than having to commit to a single style such as microprogramming or PLA. Flexibility is important because we wish to synthesize digital processors that have the proper functionality and also meet performance and cost constraints. To accomplish this, control style should vary from problem to problem. A second goal is to use existing control and logic optimization techniques. To achieve these goals, we will specify the behavior of the controller without committing to a logic implementation by creating a *Finite State Machine* specification in the form of a *state table*. The table lists the set of states, outputs, and state transitions under all input conditions. The Mealy Machine model is followed, in which outputs as well as state transitions may be conditional, that is, depend on inputs in addition to current state. Existing systems and techniques will be used for state assignment and minimization, as well as state machine synthesis.

Control specification proceeds by determining outputs and next state transitions for one state at a time, called the *current state*. The root point is taken as the start of the first state and a depth first traversal of the DDS control and timing graph is performed. Three aspects of the behavior of the current state are determined. First, the program decides which behaviors described in the graph should be included in the current state, and which point(s) of the graph signals the beginning of the next state(s). We describe the importance of arc type to this *state demarcation* process in the following section. An additional effect of point type on state demarcation will be explained within the discussion on state transition. Second, we determine which, if any, input values are tested in the current state, and how that affects the *next state transitions*. In Section 3.2 we show how point type and predicate bindings determine the conditions affecting state transition. Third, we check for current *state outputs* and the conditions under which they should be asserted. Section 3.3 describes how bindings between time intervals and values are used to determine outputs.

3.1 State Demarcation

The *arc type*, along with any associated *length specification*, is used to determine at what point of the CT graph the current state ends and a new state begins. There are four basic time interval configurations: clock period arc, clock phase arc, interval arc of unspecified length, and interval arc with a length specified. Data path synthesis assigns data operations to clock periods, which are represented in the CT graph. Not all processor behavior is associated with data path events, however. For this reason

intervals may exist in the specification that do not represent clock periods. The communication protocol used by an interface is one behavior not implemented by datapath operators. Asynchronous protocols are represented using interval arcs, with length specifications where required. If the protocol is synchronized to a clock, clock arcs are used in the CT representation. Clock phase arcs are used if signal behavior is defined in relation to the high and low phases of the clock, otherwise clock period arcs are used. Interval arcs may also be present.

A single *clock period arc* and the behavior associated with it represent one state, whereas two sequential *clock phase arcs*, representing the two phases of the clock period, are necessary to represent one state. Behaviors associated with *intervals of unspecified length* can execute in a single state, provided there is not an indefinite wait on an asynchronous event defined on the interval (the latter form is described in Section 3.2).

Two methods are used to arrive at the proper state sequence for *intervals with time specified*: a succession of individual states, or a loop consisting of a single state and a data path counter. The number of states or loop iterations is given by $\lceil length/clockperiod \rceil$.[1] Other research on control synthesis at USC has determined boundary conditions for using a loop versus a sequence of states. For a loop of length one, if the number of iterations is greater than five the most favorable implementation is a loop, whereas five iterations or fewer are most efficiently implemented as a sequence of states.

3.2 Next State Transitions

Point types and predicate bindings determine the conditions under which state outputs are asserted and state transitions occur. Point types reflect the control and sequencing relations among time intervals and their associated events. The demarcation of one state from another is determined by arc characteristics; the conditional or unconditional nature of the transition is generally determined by the types of points within the state's subgraph.

An unconditional state transition means that next state processing is unaffected by the current value of input signals and there is a single state transition from the current state to the next. This behavior is indicated by a timing graph segment that has no conditional branches. In such a graph segment all points are simple or possibly begin- or end-loop type points.

A conditional state transition occurs because the next state processing is affected by the current value of input signals. There are therefore multiple next state transitions possible; the one taken is determined by input values. We identify two timing graph configurations which are implemented by conditional state transitions, based on the synchronous or asynchronous nature of the input(s) being tested. The term synchronous is used to indicate a signal whose timing is well defined in relation to other events, though not necessarily to a predefined clock. The conditional effects of synchronous input signals are represented in DDS by an *or* point with a single incoming arc and multiple outgoing arcs corresponding to the various values of the input. The conditional value is represented by a *synchronous predicate* bound to each out-going branch. The predicate is a Boolean expression which may be composed of multiple input values.

Figure 1 shows two examples of DDS timing graphs with conditional branches and the corresponding state transitions. Referring to the top graph segment, depending on the value of the input *op*, either read or write actions, defined on subsequent arcs, are performed. We implement this in a state machine by conditional transitions from State A, during which the value of *op* is tested, to State B or State C. Multiple state transitions are necessary because we do not assume a lifetime for the input value *op* that extends beyond the interval in which it is tested. The separate states serve therefore to "remember" what the input value was. This is a necessity when branching behavior shows that subsequent behavior depends on such knowledge.

Sometimes, as shown in the bottom graph segment of the same figure, the conditional branches converge on the same point. This indicates that processor behavior varies only during a single time interval, marked State A in the figure. The output *incr* is asserted or not, depending on the value of the input *less*. Subsequent behavior is not affected. State A will therefore have a conditional output *incr* which is asserted only when the input signal *less* has the value "1" during State A. The transition under both input conditions will, however, be to the same state, B. In Figure 1, the influence of point type on state demarcation can be seen. In this example state A is defined by the behavior attached to two arcs. In general, whenever conditionals occur, multiple branches of the timing graph must be traversed to determine the behavior of single states.

Input signals that affect state transition may be asynchronous: characterized by indeterminate timing in relation to other events. Examples include reset signals and interactions between independent processors. This type of input signal behavior is represented in the DDS by *binding* the asynchronous input signal to the time interval within which it *might* be asserted, along with a point in the timing graph to which a branch is made if it is. The input is called an *asynchronous predicate* and the binding is a form of conditional branch.

We consider two categories of asynchronous input behavior[2]. The first category is asynchronous signals whose assertions interrupt processing, such as a reset. This is represented in the DDS through the hierarchy by binding an asynchronous predicate to a longer time interval composed of all shorter time intervals which can be interrupted. It is implemented by a conditional state transition based on the predicate for every state defined within the predicate's range. The second category of asynchronous input behavior is a *wait* on an input, as in the statement: `WAIT UNTIL selection-acknowledge`. The DDS representation of a simple wait (waits with time outs can also be handled) is an asynchronous predicate bound to an interval of unbounded length with no succeeding interval. This is implemented as a conditional state loop, with the state transition based on the value of

[1] The chosen clock period must be shorter than the length attached to any interval arc.

[2] Asynchronous inputs are synchronized with the controller's clock but are still indeterminate with respect to state.

the input. When the input is asserted, the loop is exited. The Multi-bus slave example of Section 4 illustrates this category of asynchronous input behavior.

3.3 State Outputs

The output of values is specified in the DDS by a binding between the value, the time interval during which it is to be output, and the structural carrier on which the value should be asserted. We know, through the state demarcation process, which arcs are associated with each state. For a particular state, therefore, we can determine the state outputs by examining the relevant arcs for bindings. All observations on the conditional and unconditional nature of state transitions made in the previous section are also relevant for determining the conditional or unconditional nature of state outputs. The only outputs with bearing upon the control specifier are those which either control data path elements or are identified with interface lines. To output data path values stored in registers, the significant bindings are those referring to the control signals of registers and multiplexers.

Those output values that have not been assigned registers or bound to control lines are I/O control signals such as are found in communication protocols. These values are defined as constants in the data flow graph, such as 0 or 1101_2. They are implemented by the controller as state outputs.

3.4 Time Constraint Satisfaction

The satisfaction of both minimum and maximum time constraints may be necessary. We implement minimum time constraints by adding *timing states* when necessary. Maximum constraints are verified after minimum constraints have been satisfied: path lengths are checked against the constraint and violators marked. Every event occurs as early as possible unless otherwise specified by minimum constraints. In particular, *procedure and subroutine calls* do not involve extra control steps for call and return events but are handled as forms of conditional transition. The logic implementation later chosen for the controller will finalize the manner of effecting such branches.

In this section we consider two factors which 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. ISYN, for example, does not analyze loops or include them in constraint satisfaction. The DDS provides a complete description of branching and loop conditions. The timing graph indicates the predicate values which lead to looping behavior. Complex predicates are defined in the data flow graph. This characteristic of the DDS allows the automatic determination of whether 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, there is limited looping behavior. If any of the three values are input from the environment, however, it cannot be determined what the number of loop iterations is. Currently it is assumed in this latter case that the number of iterations is indeterminate. Nested loops are handled provided the constructs are well-formed.

For minimum time constraints it is assumed that a loop is executed the minimum number of times. If the number of iterations is indeterminate and the loop exit is at the beginning of the loop, the minimum loop execution time is zero. If the exit is at the end, one. For maximum time constraints, all loops in the constrained path must be determinate or the timing cannot be assured and an error is reported.

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 which follow the longest branch of the conditional. Any "empty" slots which 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 notthe empty control slots are performing a timing function. This situation can lead to inefficient control design with many empty control slots. In the approach described here, states are added to paths only as needed for minimum constraints: there are no empty control states that need to be skipped over.

If properly nested, there may be multiple conditional branches. We append *timing states* to all paths which violate the minimum constraint. Whenever possible, timing states are shared between the conditional paths. Paths are arranged according to their maximal common *suffixes*, or ending state sequences. Paths may share the same timing states only if they belong to the same suffix group. The timing states are inserted just before the common suffix. This guarantees that paths that meet or exceed the minimum time will not be lengthened. Figure 2 illustrates the process with an example. There is a minimum time constraint of four clock periods between the first and last points of this timing graph. The dashed lines represent the boundaries between states. Two paths, which are marked by the symbols "*" and "+", are short and require the addition of time steps. They will not be allowed to share timing states, however, because they do not have common maximum suffixes. This assures that the path passing through p2, which meets the minimum constraint, does not have time steps added to its length.

4 Multi-bus Slave Example

The control specifier is written in Quintus Prolog and runs on a Sun 3/280 under UNIX. A simple example synthesized by ISYN in [2] was chosen to illustrate the method. In more complex examples there is less correpsondence between the DDS graph and the state diagram.

The DDS control and timing graph describing the Multi-bus slave interface is shown in Figure 3. 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←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 which 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 4), since the diagram is easier to follow. States are represented by a state label (circle): following diamonds represent the testing of input values within the state and boxes denote the outputs that are produced during the state. 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 design versus eleven in ISYN's. This example state machine was generated in 3 seconds wall clock time on a Sun 3/280.

5 Conclusions

The control specifier described in this paper has the advantage of fitting within a synthesis system that integrates the automatic design of both data path and control behavior. This extends the applicability of the method to the design of controllers for a variety of systems ranging from interface protocol transducers to data processors. We also take advantage of existing state machine and logic optimization packages to create the logic implementation. CONSPEC efficiently deals with minimum and maximum constraint satisfaction, handles internal loops, and produces a specification with multiple implementation possibilities.

Our current research is focused on adding flexibility to the control specification method in handling control/data path interactions, maximum time constraint violation, parallel branches, and clock period determination based on constraint satisfaction.

References

[1] E. Girczyc and J. Knight. An ada to standard cell hardware compiler based on graph grammars and scheduling. In *Proceedings, 1984 International Conference on Computer Design - ICCD*, pages 726–729, October 1984.

[2] J. Nestor and D. Thomas. Behavioral Synthesis with Interfaces. In *IEEE International Conference on CAD*, November 1986.

[3] G. Borriello and R. Katz. Synthesizing Transducers from Interface Specifications. In *International Conference on VLSI*, August 1987.

[4] S. Hayati, A. Parker, and J. Granacki. Representation of control and timing behavior with applications to interface synthesis. In *Proceedings of the International Conference on Computer Design*, 1988.

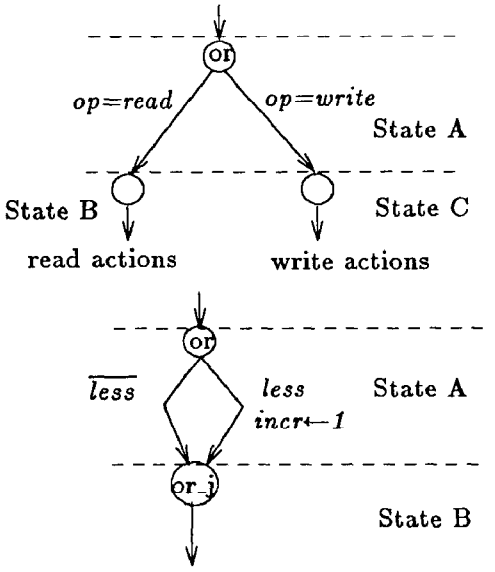

Figure 1: DDS Conditional Branches and State Transitions

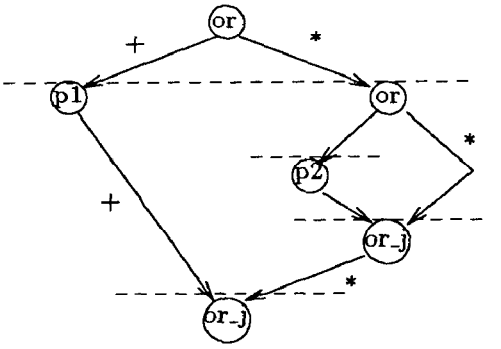

Figure 2: Conditional Paths and Minimum Timing Constraints

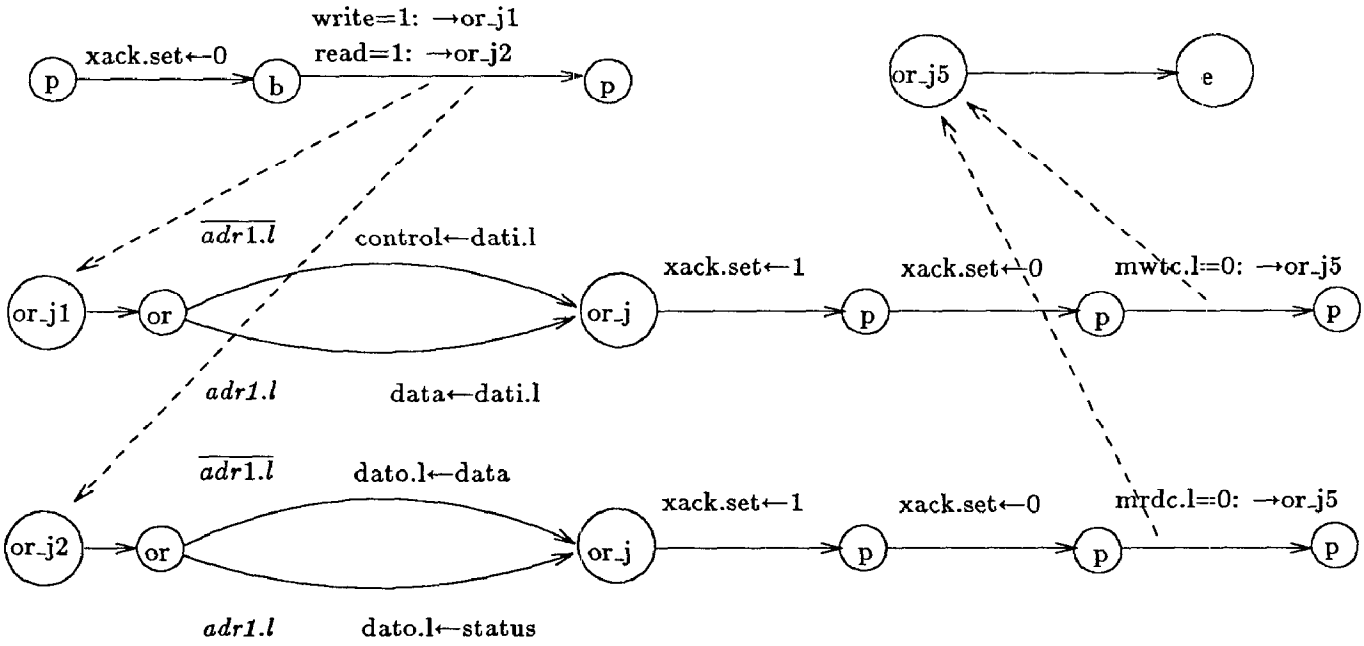

Figure 3: DDS Representation of Multibus Slave

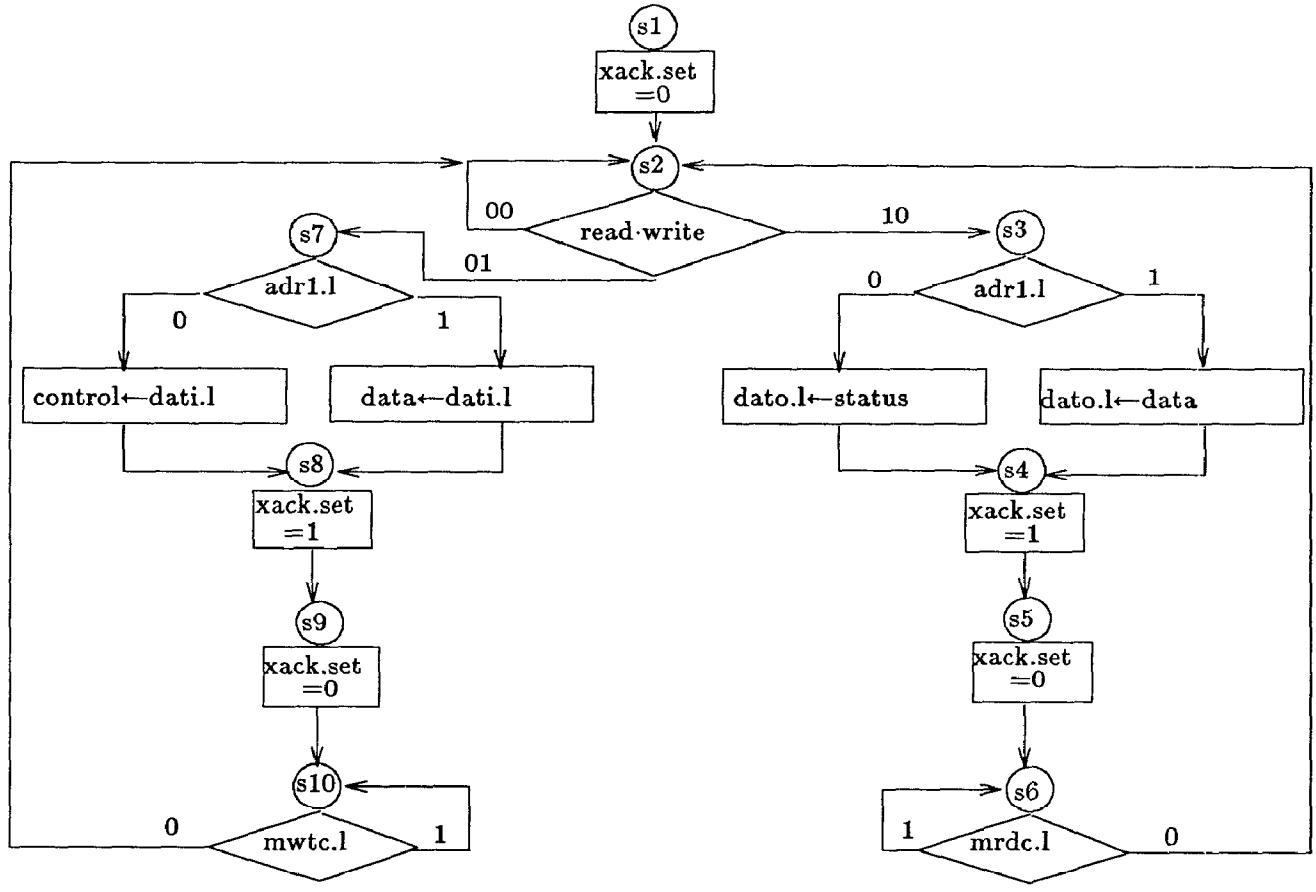

Figure 4: State Diagram Derived for Multibus Slave Example