Optimal synthesis of control logic from behavioral specifications

David C. Ku and Giovanni De Micheli

Computer Systems Laboratory, Stanford University, Stanford, CA 94305, USA.

Received November 24, 1989

Abstract. This paper presents a method of synthesizing control for synchronous digital circuits starting from a behavioral description of the hardware. The input to the control synthesis task is an abstraction of hardware behavior based on *sequencing graphs*. The model is a concise way of specifying both control and data dependencies, and support hierarchy, unbounded delay operations such as data-dependent loops and external synchronizations, and multiple threads of concurrent execution flow. We show how the sequencing graph can be mapped directly to a modular interconnection of finite state machines. The approach, called *adaptive control*, is different from other control schemes in that it takes into account the dynamic variations in the execution delay of operations due to the changing inputs. It is *optimal* by guaranteeing the minimum number of cycles in the execution of a hardware behavior for all input sequences., Specifically, there are no performance penalties for the arbitrary nesting of procedure calls, conditionals, or loops. The adaptive control model and implementation are used within the framework of the HERCULES/HEBE High-Level Synthesis system.

Keywords. Automated control synthesis, high-level synthesis, control generation

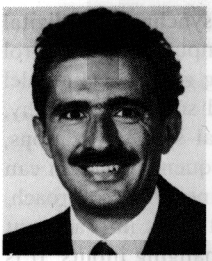
1. Introduction

High-level synthesis systems have been shown to be effective in supporting the design of Very Large Scale Integration (VLSI) digital circuits [1,5,8,14,16,17,18,21]. Numerous advantages can be achieved by designing a circuit starting from a self-documented high-level specification. In particular, the design is more porta-

Elsevier
INTEGRATION, the VLSI journal 10 (1991) 271–298

ble, incremental changes in the specification are easier to incorporate, the specification is independent of the target technology, and the design life cycle is likely to increase. Powerful synthesis techniques can optimize the circuit at the behavioral, structural, and logic levels. This results in faster design turnaround time, and increases the quality and profitability of a given design.

We have developed a high-level synthesis sytem at Stanford University called HERCULES/HEBE [7], that transforms a behavioral hardware description in the HardwareC language [10] into an implementation in terms of synchronous logic circuits. This paper deals with a specific task of high-level synthesis: the synthesis of synchronous control logic. Control synthesis is important because it affects the control flow of operations, and hence directly impacts the overall performance of the resulting hardware.



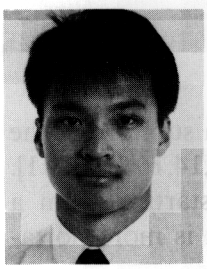
Giovanni De Micheli is Associate Professor of Electrical Engineering and, by courtesy, of Computer Science at Stanford University. From 1984 to 1986 he worked at the IBM T.J. Watson Research center, Yorktown Heights, New York, where he was project leader of the Design Automation Workstation group. Previously he held position at the Department of Electronics of the Politecnico di Milano, Italy and at Harris Semiconductor, Melbourne, Florida. He was co-director of the Advanced Study Institute on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, under the sponsorship of NATO in 1986 and in 1987.

He received a Dr. Eng. degree, *Summa cum Laude*, in Nuclear Engineering from the Politecnico di Milano, Italy, in 1979, a M.S. and a Ph.D. degree in Electrical Engineering and Computer Science from the University of California, Berkeley in 1980 and 1983, respectively.

Dr. De Micheli was granted a *Presidential Young Investigator* award in 1988. He received the 1987 *Best Paper Award* for the best paper published on the IEEE Transactions on CAD/ICAS and a *Best Paper Award* at the 20th Design Automation Conference, in June, 1983.

His research interests include several aspects of the computer-aided design of integrated circuits with particular emphasis on automated synthesis, optimization and verification of VLSI circuits. He is co-editor of the book: *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, Martinus Nijhoff Publishers, 1987.

Dr. De Micheli is a Senior Member of IEEE. He is member of the editorial board of the IEEE Design and Test magazine. He was technical and general chairman of the *International Conference on Computer Design* – ICCD in 1988 and 1989, respectively.



David Ku was born in Taipei, Taiwan, in April 1964. He received in 1986 the B.S. degree in Electrical Engineering, *Summa cum Laude*, and the B.S. degree in Computer Science, *Summa cum Laude*, both from the University of Utah. He received the M.S. degree in Electrical Engineering from Stanford University in 1987. Since 1987, he is a doctoral candidate in Electrical Engineering at Stanford University. His research interests include high-level synthesis, control synthesis, design automation and CAD frameworks. David is currently a research assistant in the Electrical Engineering Department at Stanford University.

David is a CIS/Signetics FMA Fellow in the Center for Integrated Systems at Stanford University since 1989. He was granted the AT&T fellowship in 1986. He received the *Most Outstanding Senior in Electrical Engineering* award from the University of Utah in 1986, and the *Most Outstanding Junior in Electrical Engineering* in 1985. He also received the Minority Scholastic Award in 1986, the Clyde Christensen Scholarship in 1985, the Northwestern Energy Scholarship in 1984, and the University Scholarship from 1982–1986. David is a member of IEEE and ACM.

Numerous control styles have been used in high-level synthesis systems, ranging from read-only memory (ROM) based microprogrammed controllers [21] to distributed control [1]. Although these control approaches are part of powerful synthesis systems, the issue of minimizing the execution time of the hardware behavior is not adequately explored. We present here a control synthesis approach which addresses the following three issues:

- *Support both fixed and unbounded delay operations* – Realistic hardware designs consist of operations with *fixed* delays, such as addition and multiplication, as well as operations with *unbounded* delays, such as synchronization primitives and loops with data-dependent completion. The unbounded delay operations have delays that are not known a priori.
- *Supports multiple threads of execution flow* – The power of a synthesis system lies in its ability to explore tradeoffs between serial and parallel designs, related to the tradeoffs between area and performance. For such exploration to be possible, the control model should support multiple threads of concurrent execution flow.
- *Guarantees performance optimality* – For efficient hardware design, it is important to generate a control that yields the minimum number of cycles in executing the hardware behavior for all input data sequences.

In the restrictive case where all operations have fixed and known delays, *scheduling* can be used to assign operations to specific time slots, with the control implemented accordingly [8]. In the general case, however, the synthesis of control is more complex. We present here an approach, called *adaptive control*, that directly maps an abstract hardware model in terms of sequencing graphs into a synchronous control unit consisting of a modular interconnection of interacting finite-state machines. As its name indicates, the adaptive control is able to take into account the variations in the execution times of the operations caused by the changing input data. It is *optimal* by guaranteeing execution of the hardware behavior in a minimum number of cycles for all input sequences. In particular, the hardware model derived from a procedural hardware description language incurs no performance penalties for the arbitrary nesting of procedure calls, conditionals, and loops.

The adaptive control synthesis has been implemented in the HERCULES/HEBE High-level Synthesis system and tested on many hardware design benchmarks. Since our model of hardware behavior is fairly general, this method has wide applicability to the synthesis of digital circuits. We therefore present it here as a control synthesis method based on an abstract hardware representation. This paper is organized as follows. We review first related research in the field and contrast it to our approach. We then formalize the problem in terms of a hierarchical graph representation and present two control implementations. The former is a simplified scheme that supports unbounded delays and multiple execution flows, but it does not yield optimal performance. We extend the approach in the latter control scheme to satisfy also the optimality requirement, as we prove formally. We conclude by commenting on the software implementation of this technique and on the experimental results.

2. Related research

In this section we briefly contrast our work with the related research in the area of automated control synthesis. A widely applied control style in many existing automated synthesis systems is the ROM-based microprogrammed controller model. Examples include the control allocator for the CMU-DA [15], the MCS system for AUDES [20], and the ATOMICS system for Cathedral-II [9,18]. The CMU-DA system is representative of a large number of data-path synthesis systems, including Chippe [2], Design Automation Assistant [11], Architect's Workbench [21], ADAM/MAHA [16], and HAL [17]. It assumes a canonical microprogrammed model, and performs optimizations based on the microcode format constraints. The MCS system is similar, but tailors the general microprogrammed controller model for a specific design. It performs compaction on the parallel operations and minimizes the width of the microwords using heuristic approaches. The ATOMICS system takes an RT-level description as input, and performs microprogram scheduling in order to minimize the global machine cycle count. A common assumption in these systems is that all operations have *fixed delays*. Namely, data-dependent loops and synchronization primitives such as message passing are not supported by the behavioral model.

Some systems relax the assumption on fixed delay. Among these, an effective approach is to directly map the control-data flow graph representing the hardware behavior into a corresponding state transition graph, which can be implemented by either a ROM or PLA. For example, the Yorktown Silicon compiler [1] implements control as a hierarchy of finite state machines, where a FSM is associated with each routine. Control state splitting allows tradeoffs to be made on the delay through the combinational part of the data-path. In the Karlsruhe synthesis system [5], a state transition graph is generated from the imperative portion of a description in the DSL language. The state transition graph is then optimized by merging states and physically realized. However, both systems assume a single thread of execution flow in the hardware behavior, which limits the flexibility of the synthesis system to explore architectural tradeoffs between serial and parallel design.

A novel formulation that addresses the issue of concurrency is the system proposed by Clarke [6]. A high-level specification of the control flow in a language called CSML is transformed into a state transition graph. To deal with the combinatorial state explosion encountered when dealing with concurrency of unbounded delay operations, interface modules are created for each pair of interacting state machines using a technique called compositional model checking. Unfortunately, the specification does not support data-path operation, and the state explosion problem impacts the efficiency of the resulting control implementation.

Finally, an alternate formulation of graph-based control is the system by Bruck [3]. Hardware behavior is specified using a CSP-like language called CAP/DSDL, which is mapped to a modified Petri-net model. Partitioning is performed to identify the strongly connected state machines, which can be mapped to different

implementation styles. Although it is modular and can support concurrency and external synchronization, the state explosion problem along with the duplication of subgraphs in the partitioning of Petri-nets are issues that need to be addressed.

Although the systems mentioned above are part of powerful synthesis systems, and effective in synthesizing a large class of designs, they address only part, not all, of our objectives – to support unbounded delay operations, concurrency, and to guarantee minimum execution time for all input sequences.

3. Hardware model and problem formulation

We formulate the control synthesis problem on an abstract hardware model. To justify the model, we describe first the hardware description language (HDL) from which our model originates. It is important to remark that most of the language features are common to other HDLs. We abstract these features in the *sequencing graph model* as the basis for control synthesis.

3.1. HardwareC

HardwareC is a procedural HDL with features to support interprocess communication [7]. The language models hardware as a set of concurrent and interacting *processes*. Each process represents a specific functionality that executes repeatedly. To support communication and synchronization among the concurrent processes, HardwareC has two mechanisms of process communication: parameter passing and message passing. The former model assumes the existence of a shared medium that interconnects the hardware blocks implementing the processes. Reading and writing to this medium is achieved by synchronous `read/write` operations. The latter model uses a point-wise synchronous `send/receive` message passing mechanism.

HardwareC supports the computation of arithmetic, Boolean, and relational expressions. It supports arbitrary nested *procedure calls, conditional branches and iterative loops*. `Goto` statements are not allowed. There are two variants of loops, depending on whether the iteration bound is *fixed* or *data-dependent*. Fixed iteration loops can be selectively unrolled during behavioral synthesis. Data-dependent loops include the `while-do` and `repeat-until` constructs. They

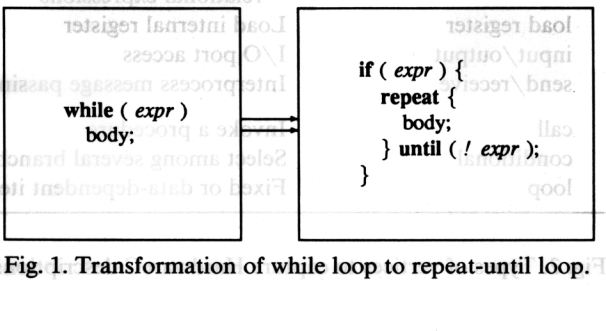


Fig. 1. Transformation of while loop to repeat-until loop.

differ in that the body of a `repeat-until` loop is always executed at least once. On the contrary, `while-do` loops may not even be entered depending on the initial value of the loop exit condition. Note that data-dependent loops may not be unrolled at compile time due to the data-dependent loop exit conditions. A `while-do` loop can be transformed into a conditional `repeat-until` loop by the transformation shown in Fig. 1. Therefore, without loss of generality we consider only `repeat-until` loops in latter discussions.

3.2. Sequencing graph abstraction

Hardware behavior described in procedureal HDL can be modeled as a set of *operations* and a *partial order* among the operations. This is represented as a polar directed graph, called the *sequencing graph*. The vertices represent the operations to be performed, and the edges represent the dependencies that are either explicit in the hardware specification, or represent dependencies due to *data-flow* restrictions or hardware *module sharing* considerations. The graph model can capture the hardware primitives of most procedural HDLs, in particular HardwareC. Variations of this model were used in [1] and [5]. The sequencing graph model differs from the Value Trace [21] in that it is primarily control-oriented.

The vertices of the sequencing graph are classified into different types according to the operations they perform. For example, the types listed in Fig. 2 are sufficient to model HardwareC descriptions [10]. The vertices are further categorized as *simple* or *complex*. The simple vertices are operations that do not involve other operations. For example, computations such as addition or Boolean expressions and message passing primitives are simple vertices. On the other hand, complex vertices allow groups of operations to be performed, and include procedure calls, conditionals, and loops. They are analogous to structured control flow constructs in most programming languages. No-op vertices are used as the source and sink vertices of the sequencing graph.

The semantic interpretation of the sequencing graph model is as follows. A vertex is *executed* by performing the task described by the vertex. For example, to execute a *computation* vertex, the task to be performed is the evaluation of the

Category	Type	Operation represented
simple	no-op	No operation
	computation	Arithmetic, Boolean, or relational expressions
	load register	Load internal register
	input/output	I/O port access
	send/receive	Interprocess message passing
complex	call	Invoke a procedure
	conditional	Select among several branches
	loop	Fixed or data-dependent iterations

Fig. 2. Types of vertices to capture HardwareC descriptions.

corresponding expression; to execute a *conditional* vertex, the operations within the selected branch are executed. In the case of a *call* vertex, the control flow is temporarily transferred to the called graph. When the called graph completes execution, the control flow returns to the calling vertex. The execution of a sequencing graph is the execution of its vertices according to the dependencies of the graph. A vertex begins execution when all its predecessors have completed execution. Since a vertex may have multiple predecessors and successors, the model supports *multiple threads of concurrent execution flow.*

The complex vertices – *call, conditional,* and *loop* – induce a hierarchical relationship among the graphs. A call vertex invokes the sequencing graph corresponding to the called procedure. A conditional vertex selects among a number of branches, each of which is modeled by a sequencing graph. A loop vertex iterates its body until the exit condition is satisfied; the body of the loop is also a sequencing graph. The sequencing graph is therefore *acyclic* because only structured control-flow constructs are assumed (no `goto`), and loops are broken through the use of hierarchy.

3.3. Execution delay

We assume in this paper a synchronous implementation of the operations and their control. Therefore we associate with each operation an *execution delay* in terms of the number of cycles required to complete its execution. For fixed delay operations, the execution delay is computed by the synthesis techniques used for the operations themselves, e.g. apply logic synthesis for delay estimates. We do not address in this paper the specification of the execution delays: we assume only that it is an integer value greater than or equal to zero. The control is synthesized based on the values of these execution delays. We can now formally define the sequencing graph model:

Definition 1. A sequencing graph $G(V, A, W)$ is a hierarchical polar weighted directed acyclic graph, where the vertices $v_i \in V$ correspond to the operations to be performed, and the directed edges $a \in A$ represent the sequencing dependencies between the operations. Each vertex $v_i \in V$ is labeled by an integer weight $w_i \in W$, $w_i \geq 0$, representing the execution delay of v_i.

A problem arises when we try to define the delay for a conditional or loop that depends on some external signal or event not known statically. To address this point, the vertices of the graph are categorized into *bounded* and *unbounded* vertices. A vertex is bounded if the time required to execute its operation is *fixed* for all input data sequences; the delay depends solely on the nature of the operation. Examples include addition and register loading. On the other hand, a vertex is unbounded if the time required to execute its operation is data-dependent. Loops whose exit condition depends on some signal value, or message passing primitives that synchronize between two concurrent processes are examples of unbounded vertices. The categorization is hierarchical. A sequencing

graph whose vertices are bounded is called a *bounded sequencing graph*. A call to a bounded sequencing graph is bounded, a conditional whose branches are all bounded is bounded, and a fixed iteration loop whose body is bounded is bounded.

The presence of unbounded vertices makes it necessary to relate the *execution delay* of a vertex *to a particular input sequence* because the time to achieve synchronization, and the number of times a loop iterates are known for a given input sequence. The execution delay of a hierarchical graph is computed bottom up, according to the following definition:

Definition 2. For a particular input sequence, the *execution delay* of a sequencing graph is equal to the length of the longest weighted path from source to sink, where:

(1) The execution delay of a call vertex is exactly equal to the execution delay of the called graph.
(2) The execution delay of a conditional vertex is equal to the execution delay of the selected branch.
(3) The execution delay of a loop vertex is equal to the number of iterations multipled by the execution delay of the loop body.

We further classify a vertex according to the value of its execution delay. A vertex whose operation requires one or more cycles to execute (execution delay > 0) is called a *state* vertex. Otherwise, it is called a *stateless* vertex (execution delay $= 0$). A sequencing graph where all the vertices are stateless is a stateless sequencing graph.

The property of *stateless / state* is independent of the classification into *bounded / unbounded* vertices. Whereas the determination of whether a vertex is bounded or not can be made *statically* by considering the type of operation it represents, the property of stateless/state may change *dynamically* as different input sequences are applied to the hardware model. The stateless/state property for bounded vertices is fixed; e.g. a *no-op* vertex is always stateless and a *load-register* (takes one cycle) is always a state vertex. For unbounded vertices, however, the property depends on the value of the execution delay for a particular input sequence. Consider for example a conditional vertex with two

Category	Type	Bounded/Unbounded	Stateless/State
simple	no-op	Bounded	Stateless
	computation	Bounded	Stateless or State
	load register	Bounded	State
	input/output	Bounded	State
	send-receive	Unbounded	State
complex	call	Bounded or Unbounded	Stateless or State
	conditional	Bounded or Unbounded	Stateless or State
	loop	Bounded or Unbounded	Stateless or State

Fig. 3. Categorization of vertices according to their properties.

branches, one state and one stateless. If for a particular input sequence the selected branch is stateless, then the conditional vertex is stateless for that input sequence. Likewise, if a new input sequence is applied to the conditional causing the other branch to be selected, then the conditional vertex becomes a state vertex. The same analysis applies to call and loop vertices. The properties are summarized in Fig. 3.

Example. To illustrate the sequencing graph model, consider the example described below in HardwareC that finds the length of a pulse (in terms of cycles). Process `length` continuously samples the input data stream, *data*, and calculates an 8-bit vector *result* as the length of the pulse. It calls the combinational procedure `Increment`, which increments its input by one. The sequencing graph for the process is given in Fig. 4. Obviously the procedure can be expanded in-line. The example serves to illustrate the use of procedures and calls.

```
process length (data, result)
   in port data;
   out port result [4];
{
   boolean counter [4];
   if (data = 0){                          /* v1 */
      repeat{                              /* v5 */
      } until (data = 1);
   }
   repeat {                                /* v2 */
      Increment(counter, counter);         /* v6 */
   } until (data = 0);
   write result = counter;                 /* v3 */
   counter = 0;                            /* v4 */
}
Increment(data, output)
   in boolean data[4];
   out boolean output[4];
{
   output = data + 1;
}
```

The execution delay for both the I/O write (v_3) and register loading (v_4) is 1 cycle, therefore they are state vertices. Since `Increment` consists of combinational logic, it is a stateless procedure.

4. Control synthesis

Given a hardware description in terms of the sequencing graph model, the task of control synthesis is to generate a control that activates the operations according to the sequencing dependencies of the graph. Our objective is to in addition

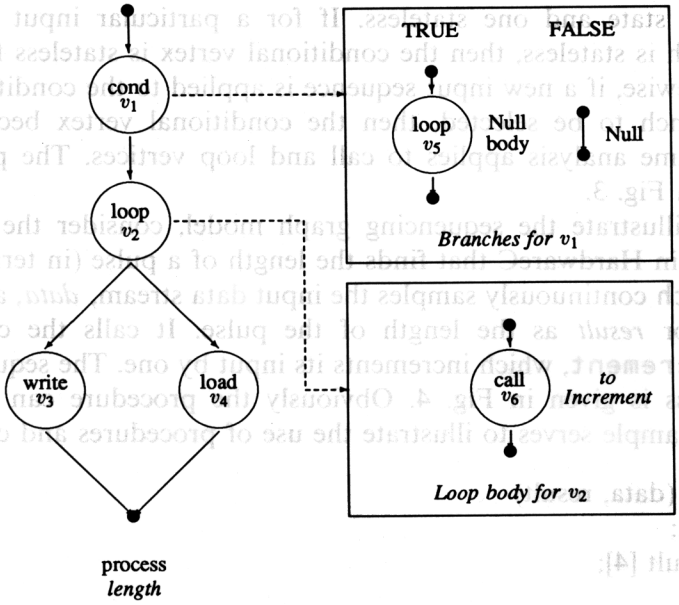


Fig. 4. Sequencing graph example for process length.

guarantee a minimum number of cycles in executing the hardware behavior for all input sequences.

To define a criterion for optimality, we need to model the control implementation of a sequencing graph. As in the case of the operations, we assume a synchronous implementation of control that can be modeled on the whole as a synchronous finite state machine (FSM), where transitions occur by the assertion of a clock signal at very *cycle* [1]. The FSM is characterized by a set of states called *control states*. For a sequencing graph, the FSM corresponding to its control implementation is assumed to be initially in the *reset state*, and returns to the reset state when all the operations of the graph have been executed. For a given input sequence, the *control delay* of the control implementation for a sequencing graph is the number of cycles to go from the reset state back to itself

The sequencing graph constrains the assignment of the operations to the FSM control states. Namely, no two states vertices connected by a path in the sequencing graph can be assigned to the same control state. Therefore the execution delay of a sequencing graph is always a lower bound for the control delay of the corresponding control implementation. We define the criterion for optimality as follows:

Definition 3. A control implementation for a sequencing graph G is *optimal* if its control delay is exactly equal to the execution delay of G for all input sequences.

[1] The model of synchronous control as a FSM serves as an abstraction to reason about its properties and it does not imply its physical realization in hardware, i.e. the control circuit may be physically implemented either as a single FSM or as a network of FSMs.

Intuitively, the time to execute an optimal control implementation of hardware behavior depends solely on the execution of the operations and not on the transfer of control. For example, if a cycle is needed to transfer control to a called procedure (as in the microcode-based implementation of [21]), then the control implementation is not optimal by the above definition. On the other hand, an optimal control implementation incurs no delay penalty in the use of control flow constructs such as procedure calls, conditionals, and loops in the HDL.

4.1. Adaptive control

We present an approach to control implementation called *adaptive control* that supports multiple threads of concurrent execution flow and the presence of unbounded delay operations. Before describing the details of adaptive control, we contrast it briefly with other control schemes that address the simplified paradigms of either fixed delay operations, or a single thread of execution flow.

In the case where all operations have *fixed delays*, scheduling techniques can be used to assign all operations to the states of a microcode sequence, as in [8,16,15]. Scheduling can be applied hierarchically to support fixed-iteration looping and conditional branching. In the case where there is a *single thread of execution flow* [1], a control automaton can be derived by assigning a state to each vertex of the sequence graph. Transitions among the states depend on the completion of the corresponding operation. The presence of multiple threads of execution flow complicates the situation. In particular, the completion of an operation is not a sufficient condition to trigger the execution of its successor because the successor may have multiple predecessors. The activation of an operator depends on the completion of execution of all its predecessors, and hence in the general case it is necessary to store the information related to the completion of an operation.

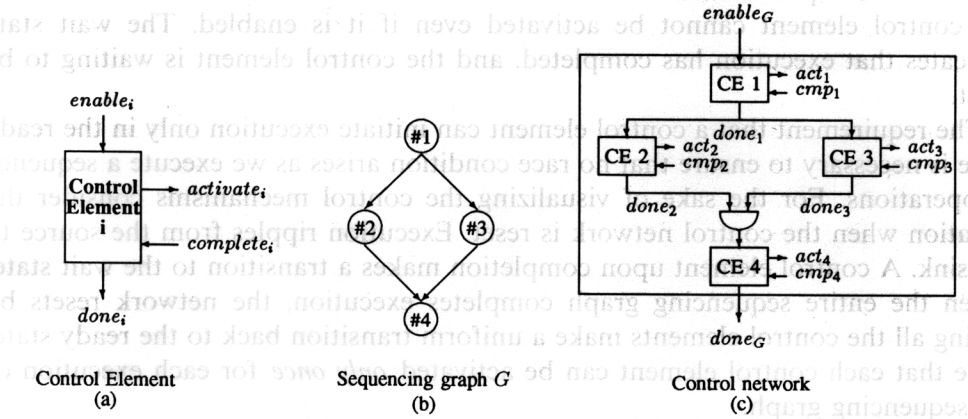


Fig. 5. Direct mapping of the sequencing graph model to the control for (a) a single vertex; (b) sequencing graph; and (c) the corresponding control network.

To support both multiple threads of execution flow and unbounded delay operations, the adaptive control is implemented as a modular *interconnection* of control elements. There is one control element per vertex of the sequencing graph, with the interconnection of the control elements having the same topology as the sequencing graph. Since there is a one-to-one correspondence between vertices and control elements, a vertex and its control element are referred to interchangeably.

To communicate among the control elements, the control element of a vertex v_i has two handshaking signals: enable_i and done_i. The *enable / done* handshaking signals indicate when a control element is enabled and when it is finished. The control element of a vertex v_i initiates its operation in the data-path using the activate_i signal, and acknowledge completion from the operation using the complete_i signal. Figure 5 illustrates the direct mapping from the graph to the control.

Control elements

The implementation of a control element is dependent on whether the corresponding vertex is *stateless* or *state*. If a vertex v_i is stateless for all input sequences (e.g., no-op or a combinatorial logic operation), then its control element asserts the done signal as soon as it is enabled. No state information is needed in this case, and the control element degenerates to the combinational logic, $\text{done}_i = \text{enable}_i$.

On the other hand, if a vertex requires one or more cycles of execution delay for some input sequences, then the control element is implemented as a synchronous FSM. The FSM has two states – *ready* (S_i^{r}) and *wait* (S_i^{w}), which can be implemented using a single bit register. The initial state for a control element is the ready state. The reset state for an entire control network is defined to be when all the control elements are in their ready states. In the ready state, a control element begins executing its operation whenever it is enabled. It remains in S_i^{r} until the completion of execution – signified by the assertion of the complete_i signal – whereupon it makes a transition to the wait state. Once in the wait state, the control element cannot be activated even if it is enabled. The wait state indicates that execution has completed, and the control element is waiting to be reset.

The requirement that a control element can initiate execution only in the ready state is necessary to ensure that no race condition arises as we execute a sequence of operations. For the sake of visualizing the control mechanisms consider the situation when the control network is reset. Execution ripples from the source to the sink. A control element upon completion makes a transition to the wait state. When the entire sequencing graph completes execution, the network resets by having all the control elements make a uniform transition back to the ready state. Note that each control element can be activated *only once* for each execution of the sequencing graph.

The adaptive control adapts to the changing execution times of the operations, and has several advantages, including modularity, distribution of control, uniform

handling of both bounded and unbounded delay operations, and support of multiple concurrent execution flows. We describe now a simple adaptive control implementation that satisfies these requirements. Although it may not be optimal in terms of execution time, we use this simple model to justify a more elaborate adaptive control scheme that is presented in the next section which satisfies the optimality requirement.

4.2. A simple adaptive control implementation

In the sequel we use the following terminology. Let $\text{pred}(v_i)$ and $\text{succ}(v_i)$ denote the set of predecessors and successors of a vertex v_i in the sequencing graph, respectively. We associate to vertex v_i the handshake signals for the corresponding control element: enable_i, done_i, activate_i, and complete_i. The handshake signals are defined below, where complete_i is generated by the corresponding operation in the data-path.

$$\text{enable}_i \equiv \prod_{k \in \text{pred}(v)} \text{done}_k \tag{1}$$

$$\text{activate}_i \equiv \begin{cases} S_i^{\text{r}} \cdot \text{enable}_i & v_i \text{ state} \\ 0 & v_i \text{ stateless} \end{cases} \tag{2}$$

$$\text{done}_i \equiv \begin{cases} S_i^{\text{r}} \cdot \text{enable}_i \cdot \text{complete}_i + S_i^{\text{w}} & v_i \text{ state} \\ \text{enable}_i & v_i \text{ stateless} \end{cases} \tag{3}$$

A vertex is enabled when all predecessors have completed execution, whereupon the corresponding operation is activated until its completion. The enable of a sequencing graph G is the enable of its source vertex, denoted by enable_G[2], likewise, the done of a sequencing graph G is the done of its sink vertex, denoted by done_G.

If a vertex v_i is stateless for all input sequences, then it asserts its done signal as soon as it is enabled. Otherwise, the control element is implemented as a two-state FSM, as shown in Fig. 6. The FSM remains in the ready state S_i^{r} until it has completed execution, after which it enters the wait state. The transition back from S_i^{w} occurs when the entire graph G has completed execution, signaled by the assertion of done_G. The transition conditions for the FSM are given below:

From S_i^{r} to S_i^{w}: $\text{enable}_i \cdot \text{complete}_i \cdot \overline{\text{done}_G}$
From S_i^{w} to S_i^{r}: done_G

Note that upon completion, the vertex remains in the ready state if done_G is asserted, corresponding to the case when the completion of the vertex results in the completion of the entire sequencing graph. Figure 7 shows the progression of state transitions, where the numbers in the vertices denote the execution delays for a particular input sequence.

[2] The enable signal of the root graph (process) of a sequencing graph hierarchy is a constant logic "1", to allow the corresponding process to restart execution upon completion.

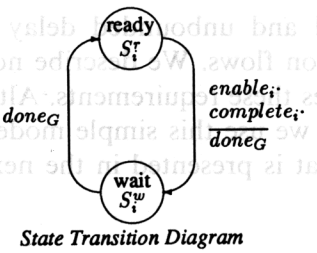


Fig. 6. State transition diagram for the simple control element.

The control element model is applicable to any type of vertices. The personalization is determined by how the control elements interconnect their *activate* and *complete* signals. For *simple* vertices, these signals are connected directly to the data-path components implementing the operations. For instance, the activate/complete signals for I/O read and write vertices are connected to the I/O ports. For load-register vertices, they are connected to the registers being loaded. For message passing send/receive vertices, the activate/complete signals are mutually coupled. Specifically, the activate of the send is connected to the complete of the corresponding receive, and the activate of the receive is connected to the complete of the send.

Additional control circuitry is needed for *complex* vertices. They are illustrated in Fig. 8, and described below:

- *Call vertex* – A network of control elements implementing a sequencing graph can be treated as a single *abstract* control element, where the enable/done of this abstract control element is connected to the enable/done of the given graph. This formulation supports hierarchy, and allows for a consistent view of the control for both a single operation and a group of operations. The activate (or

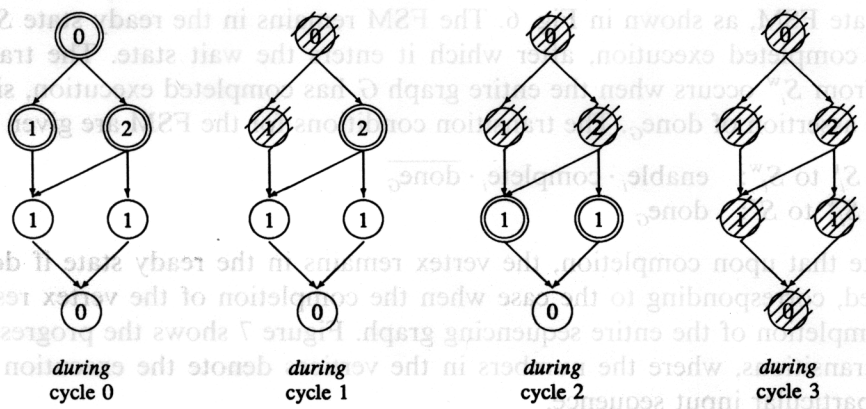


Fig. 7. The progression of state transitions in the execution of a sequencing graph. The double-circled vertices denote currently executing vertices (in S^r), shaded vertices denote vertices that have finished execution (in S^w), and unshaded single-circled vertices denote unexecuted vertices.

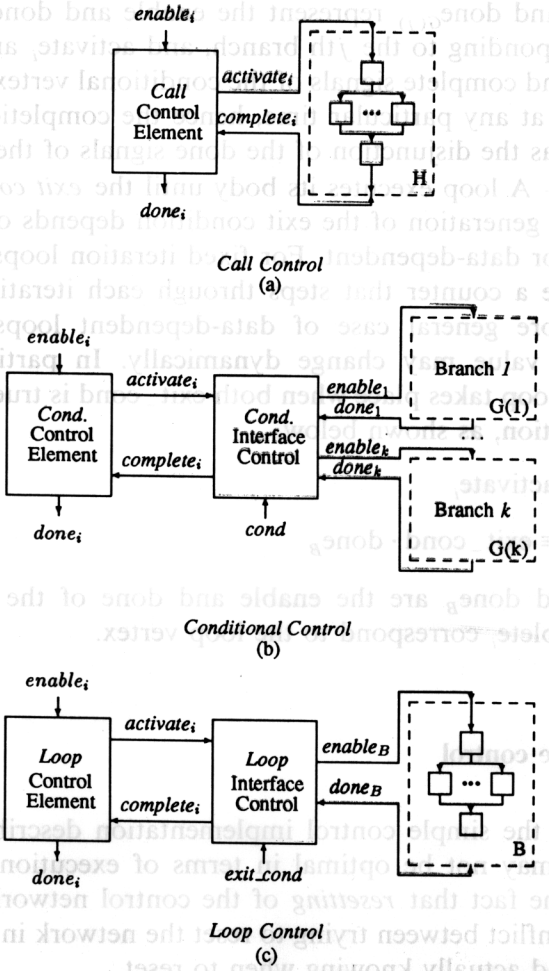


Fig. 8. Generating control for complex vertices: (a) call; (b) conditional; and (c) loop.

complete) signal of the call is connected to the enable (or done) of called sequencing graph, as shown below.

$$\text{enable}_H \equiv \text{activate}_i \tag{4}$$

$$\text{complete}_i \equiv \text{done}_H \tag{5}$$

where enable_H and done_H are the enable and done of the called graph H, respectively.

• *Conditional vertex* – Since each branch of a conditional is a separate sequencing graph, a conditional is analogous to a multi-way procedure call that depends on the value of the conditional expression. For a conditional with k branches, the handshake signals are defined as follows:

$$\text{enable}_{G(j)} \equiv \text{activate}_i \cdot (\text{cond} = j\text{th branchvalue}) \tag{6}$$

$$\text{complete}_i \equiv \sum_{j=1}^{k} \text{done}_{G(j)} \tag{7}$$

where $\text{enable}_{G(j)}$ and $\text{done}_{G(j)}$ represent the enable and done of the sequencing graph $G(j)$ corresponding to the jth branch, and activate_i and complete_i represent the activate and complete signals of the conditional vertex. One and only one branch is selected at any particular time, hence the completion of a conditional can be computed as the disjunction of the done signals of the branches [3].

- *Loop vertex* – A loop executes its body until the *exit condition* (exit_cond) becomes true. The generation of the exit condition depends on whether the loop bounds are fixed or data-dependent. For fixed iteration loops, a straightforward approach is to use a counter that steps through each iteration of the bounded loop. For the more general case of data-dependent loops, exit_cond is an expression whose value may change dynamically. In particular, exit from a `repeat-until` loop takes place when both exit_cond is true and the loop body has finished execution, as shown below.

$$\text{enable}_B \equiv \text{activate}_i \tag{8}$$

$$\text{complete}_i \equiv \text{exit\_cond} \cdot \text{done}_B \tag{9}$$

where enable_B and done_B are the enable and done of the loop body B, and activate_i and complete_i correspond to the loop vertex.

5. Optimal adaptive control

A limitation of the simple control implementation described in the previous section is that it may not be optimal in terms of execution time. An inherent difficulty lies in the fact that *resetting* of the control network occurs *too late in time*. There is a conflict between trying to reset the network in preparation for the next activation, and actually knowing when to reset.

To illustrate the difficulty, consider a sequencing graph G representing either a process or a loop body that we wish to execute repeatedly. At a particular cycle n, G is in its final cycle of execution, and done_G is asserted at the start of the next cycle $n+1$. Ideally, the control of G should be ready to restart execution during cycle $n+1$. However, this is not possible in the simple control implementation because the resetting of the control network occurs one cycle after the assertion of done_G. Consider the example in Figure 7. The control network detects the completion of the entire graph during cycle 3, and resets one cycle later in cycle 4. Therefore, the periodicity of execution is 4 cycles instead of the critical path delay of 3 cycles. This is clearly suboptimal.

When all operations have fixed delays, the restarting periodicity can be hard-coded into the control because the total delay through the graph is fixed. We

[3] We assume here that the branch condition remains unchanged for the entire duration of execution of the conditional vertex. If this assumption is not valid, then either the value of the conditional expression or the activate signals $\text{enable}_{G(j)}$ must be stored to ensure that the branch condition remains unchanged during execution of the conditional.

say in this case that the control equations can be *statically* derived. Conversely, when unbounded delay operations are present, the control equations have to consider *dynamically* the variations of the input signals. Since the total delay through the graph may change, the hard-coded control approach cannot be used. To resolve this difficulty, two mechanisms are used to achieve optimality. The first is to use lookahead to ensure proper resetting of the control for all input sequences. The second is to dynamically identify stateless operations.

5.1. Look-ahead for resetting

To ensure proper and timely reset of the control elements, it is necessary to know when a given graph is in the *last cycle* of its execution. If the detection of completion occurs during the last cycle, then the control can reset during the last cycle and restart execution in the following cycle. This guarantees the control of a sequencing graph can repeatedly execute without losing a cycle to reset, which is crucial to ensure no wasted time for restarting loops and processes.

This look-ahead requirement can be achieved by defining the complete signal of a vertex to be asserted during the last cycle of execution of the corresponding operation. For example, if an operation takes only one cycle, then the corresponding complete signal is asserted as soon as it is enabled. In general, if an operation takes n cycles to execute, the corresponding complete signal will be asserted *during* the nth cycle [4].

The redefinition of the complete signals needs to be applied consistently throughout the sequencing graph model. For simple vertices, the complete signals are generated by the synthesis system and hence pose no difficulty. However, for complex vertices that invoke other sequencing graphs, we need to detect the completion of an entire graph during the final cycle of execution, denoted by the assertion of done_G. Since a sequencing graph may have multiple threads of concurrent execution flow, we identify the set of vertices, called *direct-sink* vertices, whose combined completion results in the completion of the entire graph.

Definition 4. A vertex v of a sequencing graph $G(V, A, W)$ is a *direct-sink* vertex if the longest weighted path from v to the sink vertex, excluding the weight of v, is zero. Otherwise, the vertex is an *indirect-sink* vertex.

As an illustration, consider the sequencing graph of Fig. 9, where the numbers in the vertices represent the execution delay for a particular input sequence. The double-circled vertices are the direct-sink vertices. The completion of execution of

[4] It is straightforward to implement a control scheme that uses this look-ahead technique. The only complication is the necessity to determine whether the loop exit condition is satisfied *during* the final cycle of a loop's execution. Since the exit condition may depend on values of variables that are stored in registers and updated at the end of each iteration, the exit condition is computed with the values of variables before they are updated into the registers.

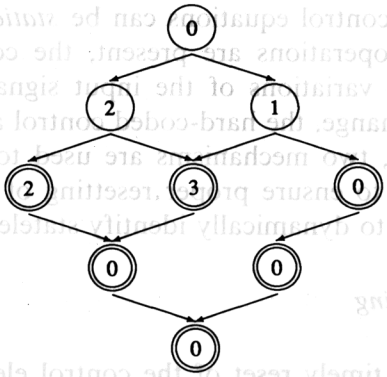


Fig. 9. Illustrating direct sink vertices. The double-circled vertices are direct sink for a given input sequence.

all direct-sink vertices results in the completion of the execution of the entire graph. Note that if the source vertex is direct-sink for a particular input sequence, then by definition the sequencing graph is stateless for that input sequence.

A complication arises since the execution delay of an unbounded vertex may change for different input sequences; in particular it may become zero, making the vertex stateless. Therefore, it is in general not possible to statically identify the set of direct-sink vertices. To address this issue, a *direct-sink* signal (dsink) is defined for each control element of the network. The dsink signal is asserted when the corresponding vertex is currently direct-sink. Specifically, the dsink_i signal for a vertex v_i is computed as follows:

$$\text{dsink}_i \equiv \prod_{s \in \text{succ}(v)} \text{dsink}_s \cdot \text{stateless}_s \tag{10}$$

where stateless_s is asserted if the corresponding vertex s is stateless. Note that dsink of the sink vertex is a constant logic "1".

With this formulation, the control can track the variations in the execution delay of a graph because the dsink signals are evaluated dynamically during hardware execution. Note that if the direct-sink vertices are all bounded vertices, then the dsink signals can be evaluated statically, not adding to the complexity of the resulting implementation.

5.2. Dynamically identify stateless computation

Computations that do not take any time to execute, such as stateless branches of conditionals, should immediately activate their successor operations without taking a cycle to transfer control. It is therefore important to know when a given operation is stateless; i.e., execution delay is zero for a particular input sequence. Furthermore, since our model supports unbounded delay operations, it is necessary to dynamically determine whether a particular operation is stateless.

We define a *stateless* signal for each control element which is asserted whenever the corresponding vertex is stateless (has zero execution delay) for a

particular input sequence. The stateless signals for simple vertices are known statically, e.g., the stateless signal for load register, synchronous I/O, or synchronous send/receive message passing vertex is always "0". The stateless signals for complex vertices are evaluated dynamically, as described below:

- *Call* – The stateless signal for a call vertex is asserted if the called graph is stateless:

$$\text{stateless}_i \equiv \text{dsink}_H \tag{11}$$

where dsink_H is the direct-sink signal of the source vertex of the called graph H, which is asserted if H is stateless.

- *Conditional* – The stateless signal is asserted whenever the selected branch is stateless:

$$\text{stateless}_i \equiv \sum_{j=1}^{k} \text{dsink}_{G(j)} \cdot (\text{cond} = j\text{th branchvalue}) \tag{12}$$

where $\text{dsink}_{G(j)}$ is the direct-sink signal of the source vertex of the sequencing graph $G(j)$ corresponding to the jth branch, and cond is the conditional expression.

- *Loop* – The stateless_i signal for a `repeat-until` loop is always "0" because the loop body executes at least once.

The use of stateless signals extends the flexibility and power of the control to adapt to the input variations. Most systems deal with loops and procedure calls in a static manner. For example, a loop in many systems takes at least one control cycle to execute, even if it is only to discover that the loop should not be entered (assuming data-dependent loops are even allowed). Likewise, a procedure is often tagged as being either *combinational* or *sequential* at compile time, corresponding to whether it requires zero or more cycles to execute, respectively. However, the implementation of a procedure may be at times combinational and at times sequential, depending on the inputs that are applied. Consider the simple description of an ALU below.

```
ALU (a, b, result, opcode)
   in boolean a [8], b [8], opcode [3];
   out boolean result [8];
{
   if (opcode = = OP_DIV) {
      / * division requiring more than one cycle * /
      result = a/b;
   }
   else
   if (opcode = = OP_AND) {
      / * simple logic operation * /
      result = a&b;
   }
```

```
    else {
      /* invalid opcode - do nothing */
    }
}
```

When the hardware modeled by ALU is executing, the null branch of the conditional is selected if the **opcode** is OP_AND, whereupon ALU becomes a combinational procedure. Otherwise, it is a sequential procedure if the opcode selects an operation requiring one or more cycles to execute, such as OP_DIV. By using the stateless signal, we can now take into consideration the dynamic variations in the time required to call a procedure, and not waste a cycle in calling stateless (combinational) procedures.

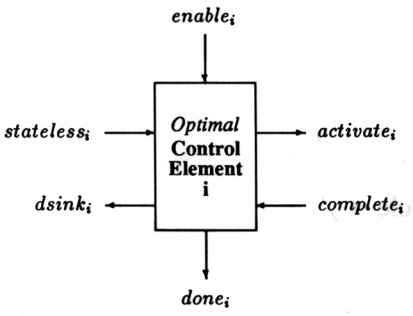
5.3. *Optimal control implementation*

The optimal control implementation extends the approach described in the previous section by incorporating the mechanisms of look-ahead resetting and dynamically identifying stateless computations. The control element for a state vertex v_i, shown in Fig. 10, has two states as in the previous model – *ready* (S_i^r) and *wait* (S_i^w). The transition conditions are now described as follows:

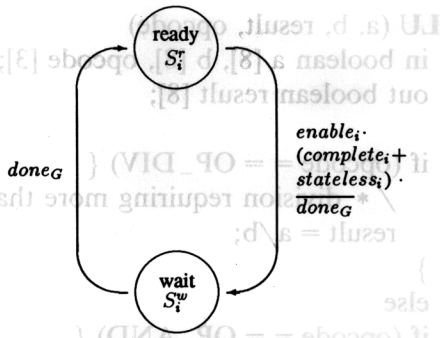
From S_i^r to S_i^w: $\text{enable} \cdot (\text{complete}_i + \text{stateless}_i) \cdot \overline{\text{done}_G}$
From S_i^w to S_i^r: done_G

The operation begins executing in the ready state. Upon completion or detection of statelessness, the FSM makes a transition to the wait state. When the entire graph is done (signified by done_G), the control element makes an uniform transition back to the ready state.

The enable signal enable_i remains the same as in the simple control implementation, equal to the conjunction of the predecessor's done signal (eqn. (1)).



Optimal Control Element
(a)



Optimal State Transition Diagram
(b)

Fig. 10. Optimal control element and its state transition diagram.

However, the activate_i and done_i are modified as follows:

$$\text{activate}_i \equiv \begin{cases} S_i^r \cdot \text{enable}_i \cdot \overline{\text{stateless}_i} & v_i \text{ state} \\ 0 & v_i \text{ stateless} \end{cases} \tag{13}$$

$$\text{done}_i \equiv \begin{cases} S_i^r \cdot \text{enable}_i \cdot (\text{stateless}_i + \text{complete}_i \cdot \text{dsink}_i) + S_i^w & v_i \text{ state} \\ \text{enable}_i & v_i \text{ stateless} \end{cases} \tag{14}$$

#	Type	Original	Reduced	FSM State
v_1	cond	$\text{enable}_1 = 1$	$\text{enable}_1 = 1$	$S_1^r \Leftrightarrow S_1^w$
	on	$\text{stateless}_1 = c_1 \cdot 0 + \overline{c_1} \cdot \text{stateless}_5$	$\text{stateless}_1 = \overline{c_1}$	
	c_1	$\text{dsink}_1 = \text{stateless}_2 \cdot \text{dsink}_2$	$\text{dsink}_1 = 0$	
		$\text{activate}_1 = S_1^r \cdot \text{enable}_1 \cdot \overline{\text{stateless}_1}$	$\text{activate}_1 = S_1^r \cdot c_1$	
		$\text{complete}_1 = \text{activate}_1 \cdot \overline{c_1} + \text{done}_5$	$\text{complete}_1 = \text{done}_5$	
		Equation 14	$\text{done}_1 = S_1^r \cdot \overline{c_1} + S_1^w$	
v_2	loop	$\text{enable}_2 = \text{done}_1$	$\text{enable}_2 = \text{done}_1$	$S_2^r \Leftrightarrow S_2^w$
	on	$\text{stateless}_2 = 0$	$\text{stateless}_2 = 0$	
	x_2	$\text{dsink}_2 = \text{stateless}_3 \cdot \text{dsink}_3$	$\text{dsink}_2 = 0$	
		$\text{activate}_2 = S_2^r \cdot \text{enable}_2 \cdot \overline{\text{stateless}_2}$	$\text{activate}_2 = S_2^r \cdot \text{enable}_2$	
		$\text{complete}_2 = \text{done}_6 \cdot x_2$	same	
		Equation 14	$\text{done}_2 = S_2^w$	
v_3	write	$\text{enable}_3 = \text{done}_2$	$\text{enable}_3 = \text{done}_2$	$S_3^r \Leftrightarrow S_3^w$
	1	$\text{stateless}_3 = 0$	$\text{stateless}_3 = 0$	
	cycle	$\text{dsink}_3 = 1$	$\text{dsink}_3 = 1$	
		$\text{activate}_3 = S_3^r \cdot \text{enable}_3 \cdot \overline{\text{stateless}_3}$	$\text{activate}_3 = S_3^r \cdot \text{enable}_3$	
		$\text{complete}_3 = 1$	same	
		Equation 14	$\text{done}_3 = S_3^r \cdot \text{enable}_3 + S_3^w$	
v_4	load	$\text{enable}_4 = \text{done}_3$	$\text{enable}_4 = \text{done}_2$	$S_4^r \Leftrightarrow S_4^w$
	1	$\text{stateless}_4 = 0$	$\text{stateless}_4 = 0$	
	cycle	$\text{dsink}_4 = 1$	$\text{dsink}_4 = 1$	
		$\text{activate}_4 = S_4^r \cdot \text{enable}_4 \cdot \overline{\text{stateless}_4}$	$\text{activate}_4 = S_4^r \cdot \text{enable}_4$	
		$\text{complete}_4 = 1$	same	
		Equation 14	$\text{done}_4 = S_4^r \cdot \text{enable}_4 + S_4^w$	
v_5	loop	$\text{enable}_5 = \text{activate}_1 \cdot c_1$	$\text{enable}_5 = \text{activate}_1$	always
	on	$\text{stateless}_5 = 0$	$\text{stateless}_5 = 0$	S_5^r
	x_5	$\text{dsink}_5 = 1$	$\text{dsink}_5 = 1$	$(S_5^r = 1)$
		$\text{activate}_5 = S_5^r \cdot \text{enable}_5 \cdot \overline{\text{stateless}_5}$	$\text{activate}_5 = \text{enable}_5$	
		$\text{complete}_5 = x_5 \cdot \text{activate}_5$	same	
		Equation 14	$\text{done}_5 = \text{enable}_5 \cdot x_5$	
v_6	call	$\text{enable}_6 = \text{activate}_2$	$\text{enable}_6 = \text{activate}_2$	always
		$\text{stateless}_6 = \text{dsink}_{\text{Increment}}$	$\text{stateless}_6 = 1$	S_6^r
		$\text{dsink}_6 = 1$	$\text{dsink}_6 = 1$	$(S_6^r = 1)$
		$\text{activate}_6 = S_6^r \cdot \text{enable}_6 \cdot \overline{\text{stateless}_6}$	$\text{activate}_6 = 0$	
		$\text{complete}_6 = \text{activate}_6$	$\text{complete}_6 = 0$	
		Equation 14	$\text{done}_6 = \text{enable}_6$	

Fig. 11. Optimized and unoptimized control for example.

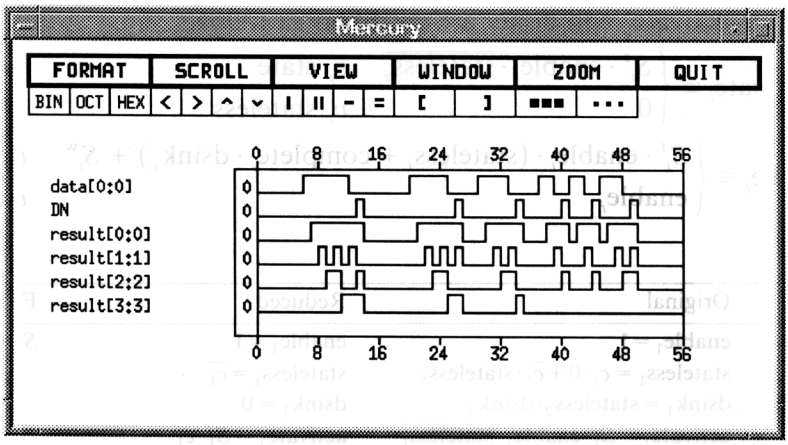


Fig. 12. Simulation result of the control for `process length`.

The activate$_i$ signal is asserted when the FSM is both enabled and in the ready state S_i^r. To explain the done signal, we note that the main difference between the implementation presented in Section 4.1 and the one described above lies in the fact that the complete$_i$ signal is asserted *during* the last cycle of execution. Therefore, the control element can detect that its operation is currently in the last cycle and makes a transition to the wait state S_i^w for the next cycle.

Example. To illustrate the application of adaptive control, consider the example of process length in Fig. 4. Figure 11 shows both the modular control implementation before optimization, and the control after variable propagation has been applied to the logic equations, e.g., $a = 1$ and $b = a \cdot c$ are equal to $b = c$. The conditional expression for v_1 is denoted by $c_1 = (\text{data} = = 0)$; the loop exit conditions for v_2 and v_5 are denoted by $x_2 = (\text{data} = = 0)$ and $x_5 = \overline{(\text{data} = = 0)}$, respectively. Note also that the loading of the counter register in the `repeat-until` loop takes place after the completion of every iteration. The simulation result for the generated control is given in Fig. 12. The value of result is delayed by one cycle because the loading of registers takes effect after the clock edge arrives.

We see that a significant amount of reduction can be achieved in the control implementation even with a simple variable propagation optimization.

5.4. Demonstrating optimality

In this section, we will prove that the implementation presented in the previous section is optimal. We begin by decomposing the definition of optimality into two criteria, both of which must be satisfied by an optimal control implementation.We say that a control element executes when it activates the corresponding operation, i.e. asserts the activate signal, and a control element completes execution when the corresponding operation completes execution. Let $\text{pred}_{\text{state}}(v_i) \subset V$ denote the

set of state vertices such that a state vertex v_p is in $\text{pred}_{\text{state}}(v_i)$ if there is a path of stateless vertices from v_p-to-v_i.

Lemma 1. Given an input data sequence, the control delay of a synchronous control implementation of a sequencing graph $G(V, A, W)$ is equal to its execution delay if the following two criteria are satisfied:

(1) *Optimal transition*: For every state vertex $v_i \in V$, the control element is enabled in the cycle after the completion of execution of all state vertices in $\text{pred}_{\text{state}}(v_i)$.

(2) *Optimal restarting*: For all polar sequencing graphs in the hierarchy induced by G, the control element of the source vertex of a given graph can be activated in the cycle following the completion of all its vertices.

Proof. Given an input data sequence, the execution delays for all operations in G are known. Therefore, the execution delay of G can be computed as the longest weighted path in G. The optimal transition criterion ensures that transferring control from any control element to its successors incurs no penalty in terms of cycles for state vertices. Recall that the done signals of stateless vertices are immediately asserted upon being enabled, and hence stateless vertices do not affect the execution delay and control delay.

The optimal restarting criterion ensures that transferring control across the hierarchy also does not incur penalties. In particular, it ensures that: (1) the control delay for a call vertex is equal to the control delay of the call graph, (2) the control delay for a conditional vertex is equal to the control delay of the graph corresponding to the selected branch, and (3) the control delay for a loop vertex is equal to the control delay of the graph corresponding to the loop body times the number of iterations. Therefore, if both criteria are met, then for a given input data sequence, the control delay for the synchronous control implementation of G is equal to the execution delay of G. □

By definition, the complete signals for simple vertices are asserted during the final cycle of execution of the corresponding operations. We show now that this is true also for the complex vertices – call, conditional, and loop.

Lemma 2. The complete_i signal for any complex vertex $v_i \in V$ in a sequencing graph $G(V, A, W)$ is asserted during the final cycle of execution of v_i.

Proof. From eqn. (5), the complete signal of a call vertex is equal to the done of the called graph. From eqn. (7), the complete signal of a conditional vertex is the disjunction of the done signals from each of the conditional branches. From eqn. (9), the complete signal of a loop vertex is the conjunction of the loop exit condition and the done of the loop body, where the loop exit condition asserts during the final cycle of execution of the loop. Therefore, it is sufficient to show that the done of a sequencing graph G is asserted during the final cycle of execution of G.

This can be shown by an inductive argument. Consider first the case where the graph G consists of only simple vertices. The done signal of G, done_G, is the conjunction of the done signals of all direct sink vertices. From the definition of dsink_i (eq. (10)) and done_i (eqn. (14)), the done signals for all direct-sink vertices v_i are asserted as soon as their corresponding complete_i signals are asserted. Since all vertices are simple, complete_i is asserted during the final cycle in the execution of v_i, and done_G is also asserted during the final cycle of execution of G.

Now consider a graph containing complex vertices, where the done signals of the graphs associated with the complex vertices are asserted during their respective final cycle of execution. By eqns. (5), (7), and (9), the complete signals of the complex vertices are asserted during the final cycle of execution. From the above argument, the done signal of the graph containing the complex vertices is also asserted in the final cycle of its execution. □

Lemma 3. The adaptive control implementation of G satisfies the optimal transition criterion.

Proof. The optimal transition criterion states that for the element of a state vertex v_i is enabled in the cycle after all the assertion of done signals from every control element in $\text{pred}_{\text{state}}(v_i)$. From eqn. (1), a control element is enabled when the done signals of all its predecessors are asserted, e.g., $\text{enable}_i = \prod_{v_p \in \text{pred}(v_i)} \text{done}_p$. If v_p is a stateless vertex, then is asserts its done signal as soon as it is enabled, e.g., $\text{done}_p = \text{enable}_p$. Therefore, the enable_i signal for v_i is asserted when the done_p signals of all $v_p \in \text{pred}_{\text{state}}(v_i)$ are asserted, i.e., $\text{enable}_i = \prod_{v_p \in \text{predstate}}(v_i) \text{done}_p$.

To prove the optimal transition criterion, it is sufficient to show that for all vertices $v_p \in \text{pred}_{\text{state}}(v_i)$, the done_p signal is asserted in the cycle after the completion of v_p. This implies that enable_i is also asserted in the cycle after the completion of all states vertices $v_p \in \text{pred}_{\text{state}}(v_i)$.

Note that since v_i is a state vertex, any vertex $v_p \in \text{pred}_{\text{state}}(v_i)$ is by definition indirect-sink vertex, e.g. $\text{stateless}_p = \text{dsink}_p = 0$. Therefore, eqn. (14) reduces to $\text{done}_p = S_p^w$. Since v_p enters the wait state S_p^w in the cycle after the assertion of complete_p, and since by Lemma 2 complete_p is asserted during the final cycle of execution of v_p, the signal done_p is asserted in the cycle after the completion of v_p. This holds for every $v_p \in \text{pred}_{\text{state}}(v_i)$, and hence the optimal transition criterion is satisfied. □

Lemma 4. The adaptive control implementation of G satisfies the optimal restarting criterion.

Proof. From Lemma 2, the done of a graph G is asserted during the final cycle of its execution, at which time all control elements make an uniform transition back to the ready states. Since a vertex can begin execution if enabled in the ready state (Equation 13), during the next cycle the source vertex can immediately begin

activation if it is enabled. Therefore, the control network is ready to restart execution in the cycle after the sink completes execution. □

We now state the optimality of the control implementation.

Theorem 1. The adaptive control implementation of a sequencing graph is optimal.

Proof. For a given input sequence, the adaptive control implementation satisfies both the optimal transition (Lemma 3) and optimal restarting (Lemma 4) criteria. By Lemma 1, the control delay is equal to the execution delay of the sequencing graph for a given input sequence.

The categorization of vertices to stateless verses state and direct-sink verses indirect-sink is dynamically evaluated through the use of the stateless and dsink signals for each control element. Therefore, Lemma 3 and Lemma 4 hold for all input sequences. Since Lemma 1 is satisfied for all input sequences, the adaptive control implementation is optimal. □

6. Implementation and practical issues

The adaptive control synthesis is implemented within the framework of the HERCULES/HEBE High Level Synthesis System. HERCULES/HEBE transforms a behavioral description of hardware in HardwareC into a synchronous logic implementation consisting of data-path and control. The data-path and control are passed to logic synthesis for combined optimizations. Logic synthesis optimizations such as MisII [1] and Minerva [7] can be applied to the control to significantly reduce the logic complexity. In this section we consider two important issues related to the practical application of adaptive control. The first issue is on reducing the area complexity of the control implementation, and the second issue is on the extraction of delay information from the combinational logic.

The direct application of the adaptive control implementation may be inefficient in terms of area due to the excessive use of registers. Consider for example a chain of n state vertices with bounded delays. The adaptive control implementation requires n registers, one for each vertex of the chain. However, if the execution delay of this chain is $m \geqslant n$, then the control can be implemented as a counter requiring $[\log_2 m]$ registers. This reduction can be generalized to concurrent chains in the graph by using *clustering* techniques to isolate maximal connected subgraphs of bounded vertices. The operations within each cluster can be scheduled, resulting in a reduction in the number of vertices and a corresponding savings in the number of registers. The adaptive control still guarantees optimality as long as each cluster has a single point of entry and exit.

An important consideration is the evaluation of the combinational logic delays in both the data-path and the control. These combinational logic delays form the

Examples	Graph Model		Control Implementation		
	# of vertices		Number of	Logic literals	
	total	state	register bits	original	minimized
up-down counter	21	6	11	230	65
traffic	7	2	3	92	24
frisc	198	130	148	1872	894
8251 main	82	18	21	946	205
8251 rcvr_sync	30	9	9	355	179
8251 rcvr_async	61	11	12	641	128
8251 xmit	80	12	14	792	183
Elliptic Filter	66	60	64	592	203
Diff-eq	41	29	33	320	133
Parker86	57	22	26	482	170
DAIO phase_decoder	58	27	35	492	187
DAIO receiver	64	16	31	528	172

Fig. 13. Results of adaptive control for several examples.

basis for the computation of execution delays that is used by the adaptive control synthesis. In HERCULES/HEBE, the combinational logic operations are first clustered into blocks, where each block is a maximally connected subgraph of combinational logic operations. A block represents the largest scope of combinational logic operations. A block represents the largest scope of combinational logic operations in the data-path within which logic synthesis techniques can be applied. Once the logic is optimized, logic synthesis extracts the critical path delay for the block. Given a cycle time T, the execution delay of a combinational logic block with *critical path delay* delay is $\lceil \text{delay}/\alpha \cdot t \rceil$ cycles. The factor α, $0 < \alpha \leqslant 1$, reflects the critical combinational delay in the control implementation. Specifically, the control equations can always be evaluated within the time $(1-\alpha)\cdot T$. Note that $\alpha = 1$ if one ignores the time to evaluate the control equations. Since the control equations are completely defined for a given graph topology, the factor α can always be found by applying logic synthesis to compute the critical delays in the control implementation.

The adaptive control approach has been used to synthesize three designs in addition to the standard benchmarks proposed by the High Level Synthesis Workshop. The first is a digital audio input output chip (DAIO) [13] that interfaces between serial data from a compact disc with the parallel microprocessor bus. The second is a discriminator circuit for the multi-anode microchannel array detector (MAMA) that consists of mainly structural interconnection of components. The third is a bidimensional discrete cosine transform chip (BDCT) for imaging applications [19]. The results of the control implementation for several examples are shown in Fig. 13.

7. Summary

This paper presents a synthesis technique for synchronous control logic. The starting point for synthesis is a behavioral hardware description that can be