Scheduling and Control Generation with Environmental Constraints Based on Automata Representations

Jerry Chih-Yuan Yang, Giovanni De Micheli, *Fellow, IEEE,* and Maurizio Damiani

***Abstract*— We introduce a framework for synthesis of behavioral models in which design information is represented using an automaton model. This model offers the advantage of supporting different constraints (e.g., timing, resource, synchronization, etc.) with a uniform formalism. The set of all feasible execution traces (schedules) is constructed and traversed using efficient BDD-based implicit state-traversal techniques.**

As an application example of this formalism, we present a novel scheduling/control-generation algorithm under environmental constraints where both the design and constraints are represented using automata. We present an algorithm that generates a minimum-latency schedule and a control unit representation. This approach is able to exploit degrees of freedom among interacting components of a multimodule system during scheduling, and is well suited for system-level design, where component encapsulation and interfacing are important.

I. Introduction

A SPECIFICATION of a circuit and/or a system at the behavioral level consists of a description of its functionality and implementation constraints. Existing high-level synthesis tools often rely on *ad-hoc* representations of constraints. Moreover, the synthesis of multimodule systems consisting of several interacting components, possibly with heterogeneous implementation styles, demands design tools to deal with nontraditional constraints that go beyond the traditional timing and resource constraints. For example, when considering scheduling independently two (or more) modules that exchange data, there are constraints due to data transfers. These constraints delimit the flexibility that can be used to optimize individual components. It is the purpose of this paper to model circuits and constraints in a uniform and efficient manner, and to present a synthesis paradigm that takes advantage of all degrees of freedom described by the model. While this method is applicable to usual high-level synthesis problems, it is particularly well suited for the design of multimodule systems, where component encapsulation and interfacing are important.

Manuscript received January 4, 1995; revised July 26, 1995. This work was supported in part by ARPA, under Contract DABT63-95-C-0049, by the ESPRIT III Basic Research Program of the EC under Contract 9072 (Project GEPPCOM), and by a Hitachi Graduate Fellowship. This paper was recommended by Associate Editor R. Camposano.

J. C.-Y. Yang and G. De Micheli are with the Computer Systems Laboratory, Stanford University, CA 94305 USA.

M. Damiani is with D.E.I., Universitá di Padova 35100 Padova, Italy.

Publisher Item Identifier S 0278-0070(96)01845-3.

1.1. Overview

This paper is mainly concerned with scheduling and control generation under constraints, e.g., timing, resource, and synchronization. Scheduling problems (e.g., under resource constraints and/or release-times/deadlines) are generally intractable [1]. Scheduling has been formulated and solved exactly as integer-linear programming problems [2], as well as using approximation techniques based on heuristics. Gajski *et al.* [3] present a survey for many graph-based scheduling techniques, most of them heuristics. A specialized instance of scheduling with synchronization constraints is dealt with in the form of interface matching in [4]. However, the restrictions placed on the communicating components prevent the algorithm from treating general synchronization constraints.

Recently, Radivojević and Brewer presented an exact method using BDD's to solve scheduling under constraints [5]. By using BDD's to uniformly capture sequencing dependencies and constraints, the method demonstrates that it is efficient for scheduling under resource and timing constraints.

However, in a system-level design environment, the approaches mentioned above lack the ability to effectively capture synchronization constraints among interacting components. Failure to do so leads to nonoptimal synthesis results because degrees of freedom between design and its environment are not fully used. For control-dominated designs, the complexity is exacerbated since the design space is more sensitive to the constraints than data-path designs. Takach *et al.* [6] presented a finite-state modeling for scheduling problems known as BFSM's. The approach allows for easy specification of timing slacks, but does not address well concurrency and synchronization issues. In addition, the manipulation of states is explicit, which may limit the size of problems that can be handled.

Our work addresses the deficiencies of previous approaches by proposing a new *representation* of the design space under constraints, and an efficient exact solution *method* to the scheduling and control generation problem. We show that modeling of system-level components is especially suitable for this framework. Our model differentiates from [7] in that we focus on automata construction from HDL models and their implicit manipulation, rather than on control-flow abstraction by means of expressions.

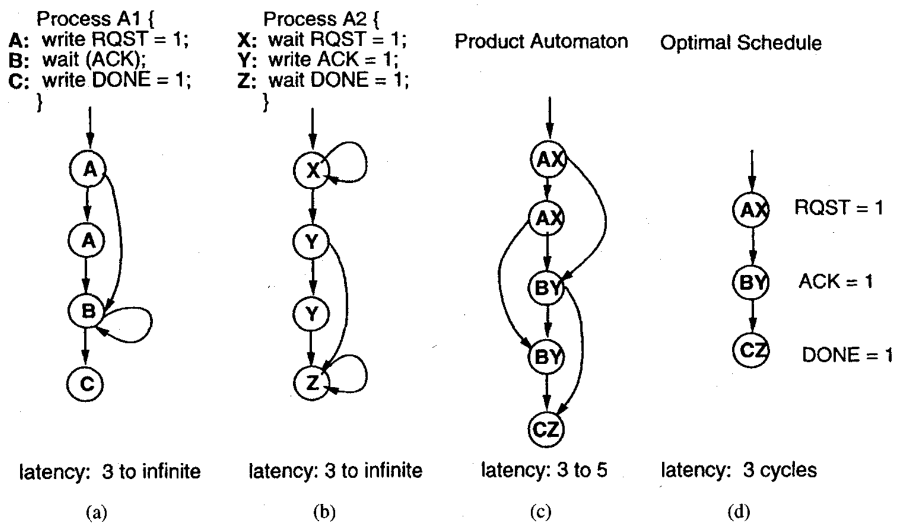


Fig. 1. Synchronization scheduling example.

The first goal of our work is a formally consistent complete representation of the control-flow design space in high-level synthesis. This representation makes it possible to derive an optimal design of control, i.e., with minimum latency. We represent the components of a design as a set of interacting components, and use notions from *trace theory* [9], [10] to capture the sequential behavior of components. Finite-state automata are used to represent these traces.

Automata modeling is able to capture design information at many levels during synthesis [11], [12]. In this paper, we use this formalism to derive a synthesis flow for control-dominated designs. The design information, as well as the constraints, are uniformly treated using finite automata as the underlying model.

An important aspect of our approach is that the automaton model is a *specification* and *synthesis* tool, which initially does not prescribe the cycle-by-cycle behavior of the circuit. Representing the set of possible execution traces, or schedules, through automata allows us to reason about the behavior of the design. In this respect, our work is different from the research focused on control-unit optimization at the logic level [13]–[15].

In order to handle efficiently the set of automata that form the design representation, we leverage results from implicit state enumeration of finite-state machines (FSM's) using BDD's. The resulting BDD model implicitly enumerates all feasible schedules that satisfy the specification and its constraints. From this model, latency minimization and control generation are performed. Retaining a *set* of feasible implementations (as opposed to picking one schedule/implementation) has the advantage of allowing additional constraints to be applied in later stages of synthesis (such as register and interconnection constraints).

The following example illustrates a typical application of our approach in finding a control solution for handshaking protocol:

Example 1: The example is illustrated in Fig. 1. Two processes $\mathcal{A}_1$ and $\mathcal{A}_2$ describing a handshaking protocol are modeled by their pseudocode and automata in Fig. 1(a) and 1(b), respectively. Each process has `wait` loops due to synchronization points. Each process also includes possible timing constraints on its operations (e.g., operation A in $\mathcal{A}_1$ and operation Y in $\mathcal{A}_2$ can occur in 1 or 2 cycles).

Therefore depending on the synchronization and timing constraints, latencies for both can be 3 cycles or more. Fig. 1(c) shows the automaton where processes $\mathcal{A}_1$ and $\mathcal{A}_2$ have been considered jointly. The automaton is obtained by taking the product of $\mathcal{A}_1$ and $\mathcal{A}_2$. Due to the exact match of synchronization conditions, the `wait` statements are effectively removed in the product, reducing the upper limit of latency from undetermined to 5 cycles. Clearly, the fastest implementation of this protocol (Fig. 1(d)) occurs in 3 cycles. This is found by taking the *shortest path* through the product automaton.

While this example may seem trivial, it illustrates an important point. It shows that without taking interactions into account, the `wait` statements cannot be easily removed and fastest implementation would not be found. □

Our approach attempts to explore the flexibility that exists in interacting components of a design. In general, our framework fits into a typical design flow as follows. First, a design is modeled using a hardware description language (HDL), e.g., *Verilog HDL, VHDL, HardwareC* [16]. The control portion of the design is then compiled into a set of interacting automata. Constraints are also represented in automata form. This is shown in Fig. 1(a) and (b) in the previous example.

An automaton representing all degrees of freedom can then be generated by taking the *product* of the constraint automata and the behavior automaton (as in Fig. 1(c)). If solutions satisfying the constraints exist, then the product automaton will be nonempty (Sections III and IV).

A minimum-latency schedule is then computed from the product automaton using a shortest path algorithm (Section VI) (as in Fig. 1(d)). We also show a simple control generation scheme using our automata model. If the design behavior contains conditionals or loops, it is often difficult to reason in terms of schedules. In those cases, instead of generating a schedule, we generate the control unit directly (Section 6.3). We show that such a control unit still executes in a minimum number of steps for any sequence of inputs. Experimental results are presented in Section VII.

II. Background

We consider the modeling and synthesis of systems that consist of interacting synchronous components. Each component runs on the same synchronized clock frequency, for the sake of simplicity. Each component is specified by an HDL model. Without loss of generality, we use *HardwareC* [16] as our specification language. The specification is compiled into a control-data flow graph (CDFG), which depicts the dependencies among operations. We use the CDFG to extract the necessary control dependencies in our modeling.

We assume hardware models can be interpreted as a set of operations and dependencies [17], [3]. Operations are assumed to take a single clock cycle to complete. Multiple-cycle operations are modeled by chains of single-cycle operations.

In our framework, we focus on control portions of synchronous, sequential digital systems. Since we consider the problems of *scheduling* and *control generation*, only portions related to control part of the behavior is needed in our modeling. The data path information is not explicitly modeled; instead we model only points of synchronization between the data path and control path.

2.1. Linking Behavior to Automata

Each component of a digital system can be modeled by using the notion of process. Process specification can be obtained by examining the ports through which a component communicates with the environment. The sequence of values that occur on the input and output ports over time can be used to specify the behavior of the component. Processes are modeled using infinite, rather than finite traces, because we assume that since the system clock is always active, there always will be a next-event. The processes we describe must be able to accept this type of "infinite sequences," also known as ω-words. A *trace* is a sequence of binary values taken over the input/output ports at each clock cycle.

Traces are represented by sequences of variables denoting operations and compositional operators. Let a and b be input symbols or expressions. The expression $a \cdot b$ denotes sequential composition of two operations, i.e., b occurs after a. The expression $a \cdots b$ denotes that b follows a and an undetermined number of operations. The alternative operator is represented by $a + b$. The expression a^* denotes the Kleene closure, i.e., *zero* or more occurrences of a. The expression a^+ is equivalent to $a \cdot a^*$, i.e., *one* or more occurrences, and a^u denotes u occurrences of a. The expression $a \otimes b$ denotes parallel composition, where the conjunction of expressions a and b is returned.

Definition 1: Let $B = \{0, 1\}$. The set of all possible infinite sequences over B is denoted by B^ω. To model a system with inputs and outputs, let $\mathcal{I}$ be the set of inputs ports, and $\mathcal{O}$ be the set of output ports. Let $\alpha = (\mathcal{I} \cup \mathcal{O})$.

A **synchronous trace** $\mathcal{T}$, or **trace** for short, is an element of the set $(B^{|\alpha|})^\omega$.

A *process* is a set of traces that describe the input–output behavior of the design. Simply put,

Definition 2: A **process** P is a set of traces (i.e., $P \subseteq (B^{|\alpha|})^\omega$).

To allow representing the languages with finite automata, we consider only ω-regular processes for the rest of the paper. The definition of process as a set of traces is another but equivalent form of expressing behavior, as compared to the notion of process in HDL's (used in Section I). We propose a deterministic automaton model (DFA) on infinite-length words as an efficient, finite representation of processes. While nondeterministic finite automata (NFA) can be more expressive in some cases [18], they do not offer any advantage for the types of acceptance conditions we deal with.

Definition 3: A *deterministic finite automaton (DFA)* $\mathcal{A}$ is defined by a five-tuple $(S, i, \Sigma, \delta, f)$, where

S finite set of **states**;
$i \in S$ **initial state**;
Σ finite set of **input symbols**, (for our purposes, $\Sigma = B^{|\alpha|}$)
$\delta : S \times \Sigma \rightarrow S$ **state transition function**;
$f \in S$ **final state.**

From a technical standpoint, the initial and final states can be merged to yield automata that model explicitly repetitive processes as defined before. The software implementation of our scheduler uses the merged notation. On the other hand, for the sake of clarity of explanation, we keep the initial and final states distinguished in the figures and examples of this article. We represent graphically an automaton $\mathcal{A}$ as a directed graph $G(V, E)$, where the set of vertices V is in one-to-one correspondence with the state set S, and the edge set E models the transitions.

We define the notion of a valid trace as follows:

Definition 4: A **successful run** of automaton $\mathcal{A}$ on a trace $\sigma = \sigma_0 \cdots \sigma_n \cdots$ is a sequence of states $s = i \cdots s_n \cdots$, such that for every $n > 0$, $s_{n+1} = \delta(s_n, \sigma_n)$.

An input sequence σ is **accepted** and called a **valid trace** if it has a successful run on $\mathcal{A}$.

Other types of acceptance conditions are reported in the literature on automata on infinite words (for example, in [18]). These rules usually require infinite traversal of some state or edge set in order for a sequence to be successful runs. Such conditions are used in verification contexts to describe **liveness** properties of processes and are not considered in the synthesis aspects of this work. For this reason, we consider only the automata and acceptance rules just defined. The traditional product rules for ordinary automata is sufficient [19]:

Definition 5: Given DFA's $\mathcal{A}_1, \cdots, \mathcal{A}_k$ over a common alphabet Σ, we define their *product* $\mathcal{A} = \otimes_{j=1}^{k} \mathcal{A}_i$ to be a DFA over Σ with the following properties:

- **State space:** $S = \otimes_{j=1}^{k} S_i$
- **Initial state:** $i = \otimes_{j=1}^{k} i_j$
- **Final state:** $f = \otimes_{j=1}^{k} f_j$
- **Transition function:** $\delta = \otimes_{j=1}^{k} \delta_j$

where $\otimes$ is the Cartesian product.

2.2. *Implicit Product Computation*

In our model, the constraints correspond to restrictions placed on the set of execution traces and are modeled by automata as well. The product of all automata modeling the digital system and its constraints describes the trace set modeling valid implementations. The technique for computing the product automaton is the well-known BDD-based technique which computes the product automaton using an implicit breadth-first traversal [20], [21].

Here we present a brief review: the underlying operator used is the *image* operator, which computes states reachable in one transition given input, present state, and the transition function:

Definition 6: Let $\mathbf{i}$ denote the set of input variables, $\mathbf{x}$ denote the present-state variable set, and $\mathbf{y}$ denote the next-state variable set. The state transition relation is denoted by $\mathbf{y} = \delta(\mathbf{i}, \mathbf{x})$.

States are represented by an encoding. We will denote by $s_k(\cdot)$ the encoding of one or more states. In particular, s_0 denotes the encoding of the initial state i.

For notational simplicity, we define the **characteristic equation** χ of the transition relation to be $\chi(\mathbf{i}, \mathbf{x}, \mathbf{y}) = 1$.[1].

In the traversal, the **next state set** s_{k+1} is computed by

$$s_{k+1} = \mathbf{Img}(s_k) = \exists_{\mathbf{i},\mathbf{x}}(\chi(\mathbf{i}, \mathbf{x}, \mathbf{y}) \cdot s_k(\mathbf{x}))$$

where $\exists_i(f) = f_{i=0} + f_{i=1}$ (existential quantification). **Img** is the **image** operator.

The **previous state set** is computed by:

$$s_{k-1} = \mathbf{PreImg}(s_k) = \exists_{\mathbf{i},\mathbf{y}}(\chi(\mathbf{i}, \mathbf{x}, \mathbf{y}) \cdot s_k(\mathbf{y}))$$

where **PreImg** is the **inverse image** operator.

In our framework, the overall design and constraint space is defined by the set of automata $\mathcal{A}_j$, $j = 1, 2, \cdots, m$. We utilize the implicit procedure to find the product by defining:

- Initial state set:

$$s_0 = \prod_{j=1}^{m} i_j$$

  where i_j is the initial state of $\mathcal{A}_j$.
- Transition relation:

$$\chi = \prod_{j=1}^{m} \chi_j$$

  where χ_j models implicitly the transition relation for $\mathcal{A}_j$.

We introduce a *restriction function* $\mathcal{R}$ which is used to specify any additional conditions that may result during the formation of the automata. $\mathcal{R}$ is a Boolean function that is used to further restrict χ. When the restriction function is used, the **next state** computation is defined as

$$s_{k+1} = \mathbf{Img}(s_k) = \exists_{\mathbf{i},\mathbf{x}}(\chi(\mathbf{i}, \mathbf{x}, \mathbf{y}) \cdot s_k(\mathbf{x}) \cdot \mathcal{R}(\mathbf{x}, \mathbf{y})).$$

Efficient implementations, such as the implicit enumeration technique, are essential to forming the product automaton with no size explosion.

[1]In practice, **i**, **x**, **y** are Boolean-encoded representations of the initial, present, and next states, respectively.

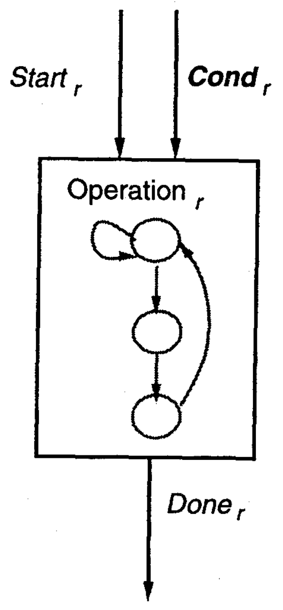


Fig. 2. Input symbols for an operation.

III. Modeling Basic Control Flow with Automata

We model hardware as a set of concurrent processes, with environmental constraints of several types (e.g., timing, resource, communication) which represent constraints on the set of acceptable execution sequences. In this section, we show how we map a control flow in terms of a set of automata. While we use an explicit automata notation for the sake of explanation, the actual manipulation of automata is done in the implicit manner described in Section II.

The control flow behavior is modeled by defining automata representations for several elementary *control constructs*. The HDL description is first translated into a structured, intermediate format such as a CDFG. Behavioral transformations (such as code-motion optimization, dead code elimination) can be applied at the CDFG level. The (optimized) CDFG flow is then analyzed and mapped into automata constructs that model elementary control constructs described in this section. Currently, hierarchies in the description are flattened.

We view behavior as a set of interacting components, where each component is modeled by an accepting automaton. First, we state the timing assumption for automata:

Assumption 1— Timing Semantics for Automaton: One **clock cycle** expires each time a **transition edge** is taken. No time parameters are associated with vertices.

The set of input symbols Σ can be separated into two sets: *control* signals and *conditional* signals. The control signals define the start and end of execution for an operation. The conditional signals occur in conditionals and data-dependent loops, where run-time values determine the execution of control flow.

Definition 7: The **input symbols** are defined with respect to interaction among operations. For an operation r, the input symbol set consists of:

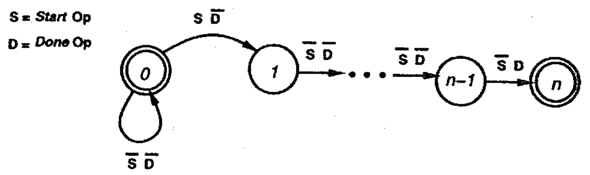


Fig. 3. Automaton for an operation consuming n cycles.

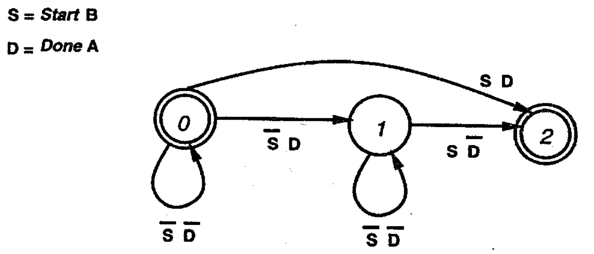


Fig. 4. Automaton for sequencing between operations A and B.

- **Control inputs**: $Start_r$ and $Done_r$ are signals that are true at the start and finish of operation r.
- **Conditional inputs**: If execution of *op* depends on data-inputs, then $\mathbf{Cond}_r$ denotes the **set** of conditional inputs for r.

Since there may be more than one conditional signal for operation r, we use the vector notation $\mathbf{Cond}_r = \{Cond_{rs}, s = 1, 2, \cdots, |\mathbf{Cond}_r|\}$. Note that vectors are emboldened.

These signals are illustrated in Fig. 2. For conciseness, we use $S_r = Start_r$, $D_r = Done_r$, $\mathbf{C}_r = \mathbf{Cond}_r$, and S, D, $\mathbf{C}$ where the operation is uniquely identified

Notationally, an automaton is represented by an expression whose supports are literals over S_r, D_r and $\mathbf{C}_r$. A minterm in such an expression $(S_r D_r C_{rs})$ represents the simultaneous occurrences of input symbols S_r, D_r, and C_{rs} on the input.

3.1. Operations

The elements of a control flow to be scheduled are called *operations*. These typically include arithmetic, logic, and I/O operations.

Definition 8: An **operation automaton** for operation A with n-cycles delay is defined by

$$Op_A(n) = ((\overline{S_A}\,\overline{D_A})^* + ((S_A\overline{D_A})\cdot(\overline{S_A}\,\overline{D_A})^{(n-2)}\cdot(\overline{S_A}D_A)))^{\omega}.$$

Fig. 3 illustrates the **operation automaton** for operation A that can consume n cycles (i.e., a multicycle operation).

3.2. Sequencing Operator

The sequencing operator is the fundamental construct that enforces the serialized execution of two operations (e.g., B executes after the completion of A). This automaton describes the behavior between the *Done* symbol of one operation (A) and the *Start* symbol of the subsequent operation (B). Fig. 4 illustrates the **sequencing automaton**.

Definition 9: A **sequencing automaton** enforces that an operation B can only start execution *some time* after A completes. This is denoted by

$$Seq_{AB} = ((\overline{D_A}\,\overline{S_B})^* + (D_A S_B)^* + ((D_A\overline{S_B})\cdot(\overline{D_A}\,\overline{S_B})^* \cdot(\overline{D_A}S_B)))^{\omega}.$$

The transition from A to B in general can take anywhere from 0 to ∞ cycles. In the next section, we show how specific timing constraints can be placed on sequencing automata.

Example 2: In this example, we show a graphical simulation of the automata executing a simple sequence of operations. Given two sequential, 1-cycle operations A and B, we show the states of the automata (dark states are active) as the signals $Start_A$, $Done_A$, $Start_B$, $Done_B$ become valid. The overall product automaton is $Op_A \otimes Seq_{AB} \otimes Op_B$.

In this example, the transition from A to B takes 1-cycle, although in general it is possible to have a 0-cycle transition (i.e., $Done_A = Start_B = 1$).

Note that it requires 4 cycles to complete a 3-cycle sequence. This is due to each operation containing a *Start–Done* signal pair, which requires two transitions to complete. Therefore, the last *Done* transition ($Done_B$ in this case) is ignored when computing latency of the model. □

3.3. Parallel Execution of Operations

In our model, operations are executing in parallel, unless there are sequential constraints. We use the following example to illustrate operation and sequencing automata and the role of concurrency.

Example 3— Addition Example: The following segment specifies a simple execution of loading, adding, and storing some values.

```
Add_Process (A, B, C, X, Y)
in port A, B, C;
out port X, Y;
{
  H: load A ;
  I: load B ;
  J: load C ;
  K: X = A + B;
  L: Y = A + C;
  M: store X;
  N: store Y;
}
```

We make some simplifying assumptions that there is only one input port and one output port. The **load** and **store** statements are executed in sequence. We also make the assumption that all operations require one cycle to complete.

A CDFG annotated with the set of interacting automata for this segment is shown in Fig. 6. The gray ovals indicate *Op* automata, and arrows indicate *Seq* automata. The set of acceptable traces for the execution of the HDL model is determined by the product automaton, which exhibits all possible scheduling of operations. The segment first sequentially loads values for A, B, and C. The sequencing of **load** and **store**

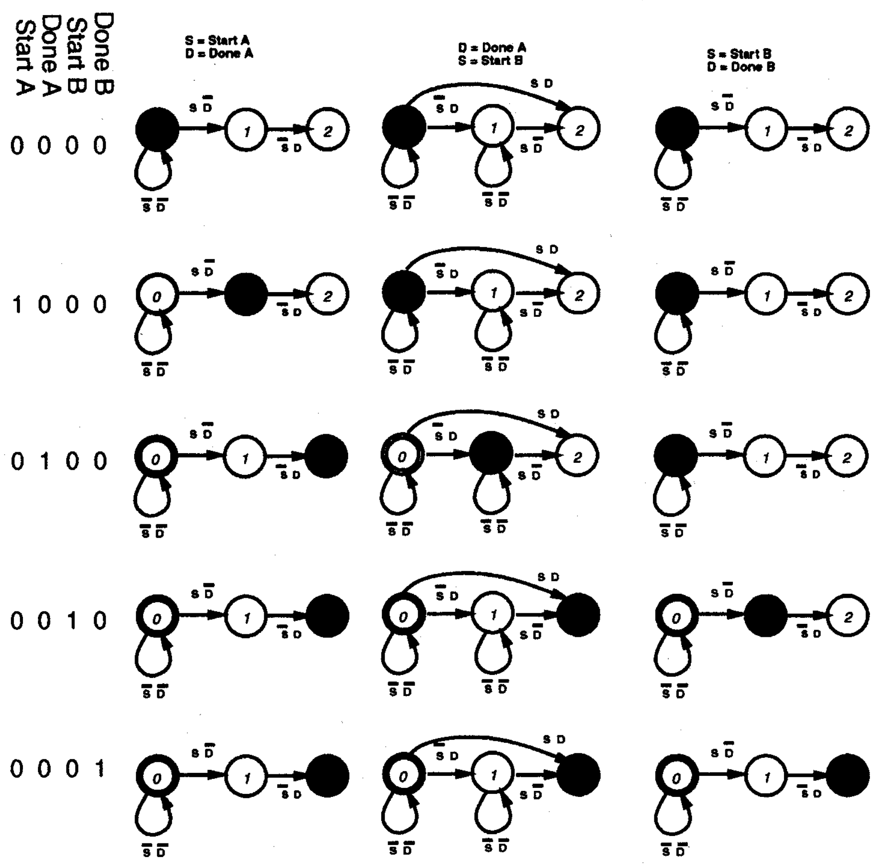


Fig. 5. Simulation of automata modeling operation A, the sequencing of $A \rightarrow B$ and operation B.

operations are fixed. The different orders in which addition can be performed is shown in the product automaton.

We have taken the operations in the dotted box of Fig. 6, and computed the product automaton, shown in Fig. 7. State s_1 indicates the starting state of this portion of the computation.

The additions $X = A+B$ and $Y = A+C$ can be performed in one of following ways.

1) The value of X and Y being computed concurrently is indicated by the sequence of states (s_1, s_4, s_8) (one cycle in doing the additions).
2) The value of X being computed before Y is indicated by the sequence of states (s_1, s_6, s_7, s_8) (two cycles in doing the additions).
3) The value of Y being computed before X is indicated by the sequence of states (s_1, s_3, s_2, s_8) (two cycles in doing the additions).
4) The automaton also illustrates the case where the computations of X and Y are delayed. The sequence $(s_1, s_5, \cdots, s_5)$ shows that the operations can wait any number of cycles before proceeding to (s_3, s_2, s_8), (s_6, s_7, s_8), or (s_4, s_8).

Therefore, depending on the resource constraints on the adders, or the timing constraints among operations, the product automaton provides the flexibility in scheduling the control flow operations. □

3.4. Conditional Operations

A key component in describing control flow is the modeling of *alternative* execution flows, conditioned by the result of some test.

A sample conditional code segment is

```
A;
if (C)
 X;
else
 Y;
D
```

In this framework, conditional constructs are mapped into automata as follows. Let us refer to the previous fragment of code. Operation X is executed if the value of a Boolean variable C is 1; else operation Y is executed. In order to define completely the execution, it is important to define **the time** at which the value of C is sampled. We chose to make this time coincident with the completion of operation A. This is not, however, the only possible choice.

Fig. 8(a) shows the CDFG relative to the fragment of code. In the original CDFG, the two operations X and Y are hidden under the `if` node by the hierarchy of the description. During the compilation of the CDFG into the automaton, we flatten the graph as shown in Fig. 8(b). The triangular vertices are called *fork* and *join*, respectively.

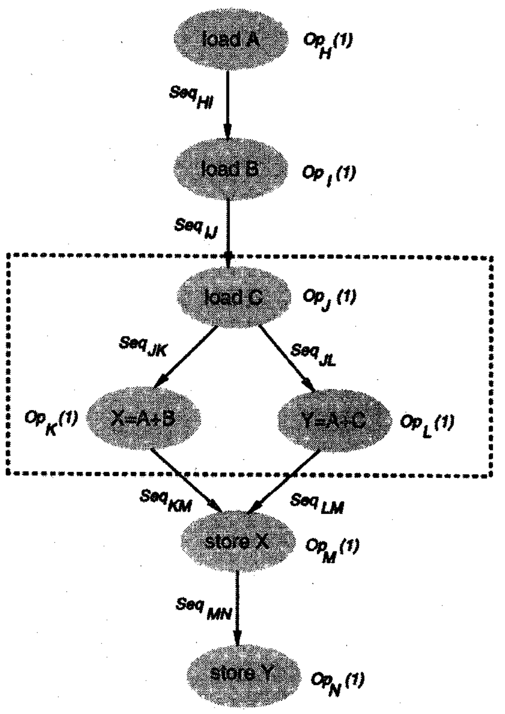


Fig. 6. CDFG/Automata notation for a segment of addition example.

The expansion operation corresponds to a rewrite of some of the $Start$ and $Done$ signals involved. The rewriting rules in this case are the following:

$$\begin{aligned} &Fork\ node: \\ &\qquad For\ node\ X: Done_A \rightarrow Done_A \cdot C \\ &\qquad For\ node\ Y: Done_A \rightarrow Done_A \cdot \overline{C} \\ &Join\ node: \\ &\qquad For\ node\ D: Done_{if} \rightarrow Done_X + Done_Y \end{aligned}$$

and are shown in Fig. 8(c).

A different choice of the sampling time for C may lead to different rewrite rules, or to the need of constructing a new sequencing automaton for X and Y altogether.

We allow `if` statements to be replaced by more general `case` statements, where only one branch is taken. In general, the completion of join is specified by the *disjunction* of all $Done_i$ signals, where $Done_i$ is the completion signal of the i^{th} branch in the conditional.

Since paths in the CDFG model are executed according to the value of the conditionals, some sets of paths have mutually-exclusive execution. Different models for enforcing mutual exclusiveness, such as *condition vectors* by Wakabayashi *et al.* [22] and path-based synthesis by Camposano [8], have been used in the past. In our formalism, mutually exclusive paths in the CDFG correspond to invalid execution traces and unreachable states in the final automaton. By the definitions

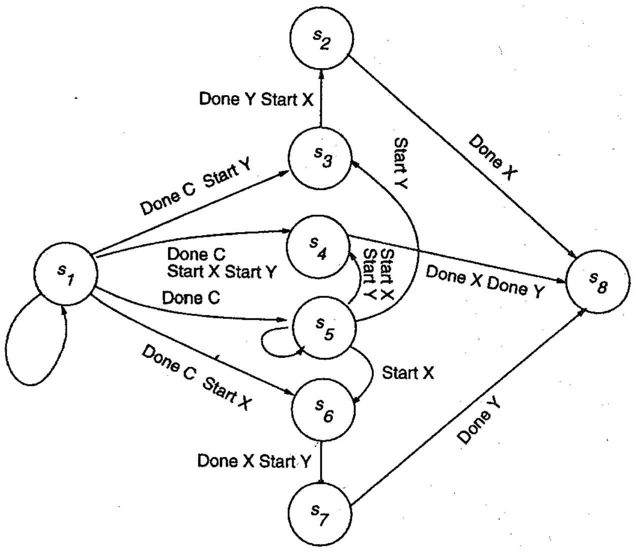


Fig. 7. Product automaton for addition section.

of *fork* and *join*, we guarantee that operations that can be executed concurrently are mutually compatible. We conclude this section with one more example, that clarifies the role of the sampling time of the condition signal C.

Example 4: Consider a CDFG as shown in Fig. 9. The triangles indicate fork and join nodes, and circles indicate operations. If C is true, then operations W and Y execute, otherwise operations X and Z execute. Thus, the sets $\{W, Y\}$ and $\{X, Z\}$ are mutually exclusive.

The automata model corresponding to the CDFG is specified by the product of all automata modeling the conditionals and operations. Namely, for the conditionals:

$$For\ X: D_U \rightarrow D_U \cdot \overline{C}$$

$$For\ Y: D_U \rightarrow D_U \cdot C$$

$$For\ Z: D_U \rightarrow D_U \cdot \overline{C}$$

$$For\ W: D_U \rightarrow D_U \cdot C.$$

The execution of operation V is subject to the termination of **both** conditionals. One automaton is constructed for each sequencing constraint, and the two D_{if} signals are replaced by:

$$D_W + D_X$$

$$D_Y + D_Z.$$

Assuming that operation U has completed ($D_U = 1$), the set of next states is the image of the current states under transition relation for any value of the conditional C. The next states contain **two** sets of states, namely, those reached with $C = 0$, and those reached with $C = 1$. By checking the operation sequencing automata, however, and recalling the rewriting occurred, it follows that the first set contains only those states in which processes X and Z are allowed to execute ($Start_X$

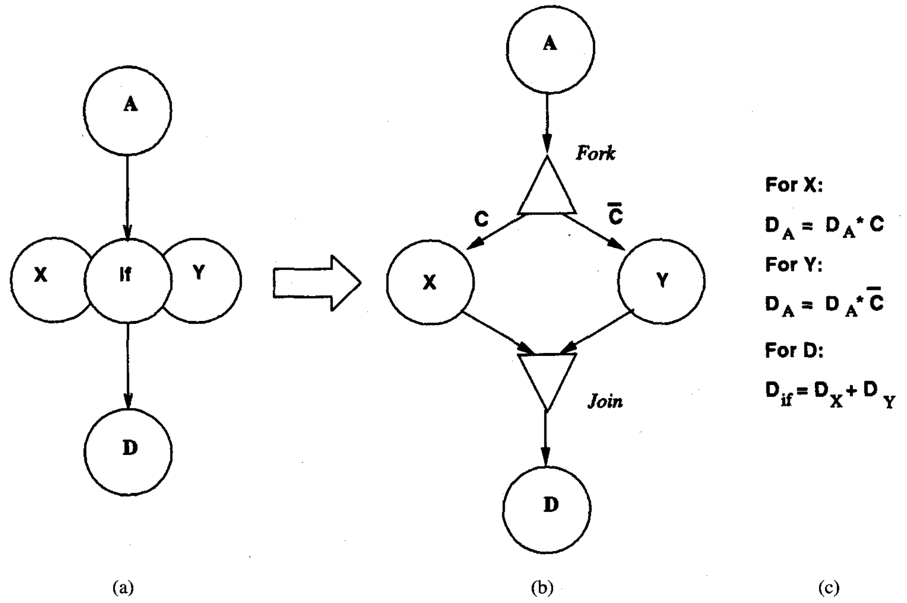


Fig. 8. Handling of conditionals: (a) Conditional branching CDFG, (b) Expansion the CDFG, (c) Rewrite rules.

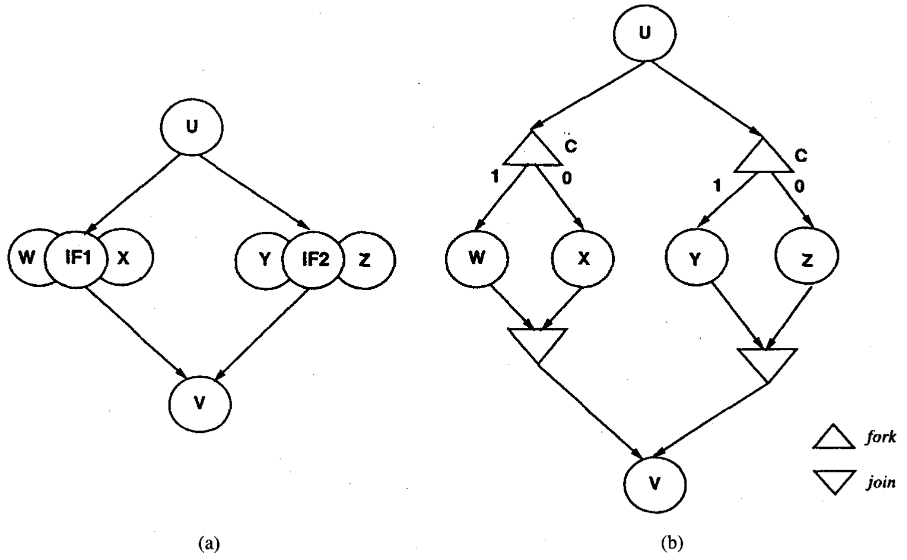


Fig. 9. Handling of parallel conditionals. (a) CDFG. (b) Expansion of CDFG.

and $Start_Y$ can be set to 1) , while in the second set only W and Y can execute. Executing other subsets of operations in $\{W, X, Y, Z\}$ is not represented by any trace accepted by the product automaton, and therefore is not allowed. Notice that the value of C used in the rewrite rules is sampled at the same time-point for all branches, namely, when $D_U = 1$. Transition in the sequencing automata for $\{W, X, Y, Z\}$ are thus synchronized to the same value. Different rewrite rules may result in different patterns of execution, because the value of C may be sampled at different times for different branches.

We conclude by observing that the parallel sampling of C for the two branches was enforced by a common parent node. If a designer wishes nonadjacent conditionals to sample the same signal, an explicit synchronization (or latching of the sampled value) must be imposed. □

3.5. Loop Execution

Loops can be classified as having fixed (known at compile time) or variable iteration count (e.g., synchronization `wait` statements, data-dependent loops). They can be described in our framework using constructs previously described. In the following example, we model a variable-iteration loop.

Example 5: A generic loop has the following form:

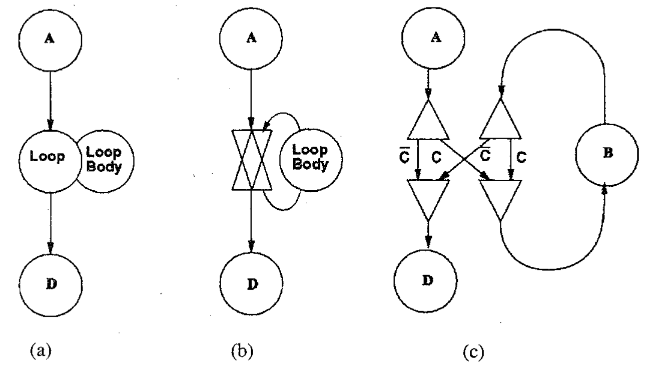


Fig. 10. Loop modeling: (a) Iteration in CDFG. (b) Symbol for iteration. (c) Expansion of iteration with conditionals.

```
A ;
while (C) {
 B = (modify C);
}
D
```

We assume C to be a Boolean variable (for instance, the result of a comparison operation). The notation used to model the loop is shown in Fig. 10. In the figure, the following execution paths are included:

- A to D (C is false initially);
- A to B (C is true, entering loop);
- B to B (C is true, already in loop);
- B to D (C is false, exiting loop).

Depending on the value of C, completion of A can start the execution of B or D. Then, B executes until C fails. Overall, the rewrite rules yield

$$For\ D : Done_{loop} \rightarrow Done_A \cdot \overline{C} + Done_{Body} \cdot \overline{C}$$
$$For\ Body : Done_A \rightarrow Done_A \cdot C + Done_{Body} \cdot C.$$

□

As in the case of conditional branching, it is important to specify the time at which the loop exit condition is sampled. We sample this condition at the end of the execution of the operation immediately preceding the loop test, which can be either the operation preceding the iteration statement or the last operation in the loop body.

The implicit traversal formalism deals with loops elegantly since traversal through a loop ends when no new states are reached. Loops are modeled by a finite-state control structure (even when the loop itself can have *infinite* latency), and can be handled conveniently using our framework.

IV. Modeling Conventional Constraints with Automata

In this section, we show how constraints are modeled. Constraints are themselves represented in automata form, and incorporated into the specification automaton by forming the *product* automaton.

In cases where there is no solution that satisfies the constraints, the product is a *null automaton.* Intuitively, this means

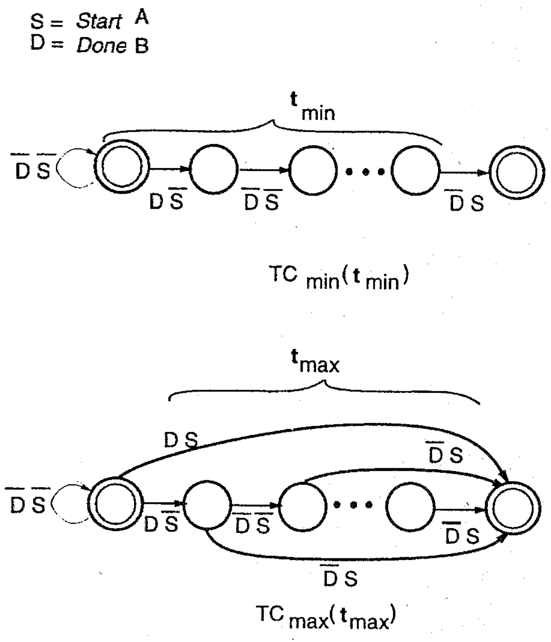


Fig. 11. (top) Automaton for minimum timing constraints. (bottom) Automaton for maximum timing constraints.

that there is no intersection between the design space and the constraint space. Conversely, as long as the product is not null, there exists at least one sequence of operations from which a controller can be synthesized.

Although the concepts in this paper are described in explicit notation, note that the actual manipulation of automata is done in the implicit manner described in Section II.

4.1. Timing Constraints

Fig. 11 shows the two general automata models for timing constraints.

To model a minimum timing constraint (Fig. 11(top)) of t_{min} cycles between two operations A and B, the constraint automaton contains t_{min} sequential transitions to NOOP states, and a self-loop transition on the initial state. The automaton is denoted by $TMIN_{AB}(t_{min})$. The automaton forces at least t_{min} cycles to expire before final state is reached.

For a maximum timing constraint of t_{max} between A and B (Fig. 11(bottom)), denoted by $TMAX_{AB}(t_{max})$, each intermediate NOOP state contains a transition to the final state. Operation B therefore must be reached by *at most* t_{max} cycles.

Example 6: (Timing Constraint) Consider the addition computation of Example (3), and suppose we wish to impose a *maximum* timing constraint of 3 cycles between **load** A and **load** B. In the automaton model, the Seq_{AB} automaton is replaced with a $TMAX_{AB}(3)$. Thus, the transition now must take place in three cycles or less. This is shown in Fig. 12(a).

A *minimum* timing constraint of 2 cycles between **load** B and **load** C. Similarly, the Seq_{BC} automaton is replaced with $TMIN_{BC}(2)$ as shown in Fig. 12(b). The automaton models

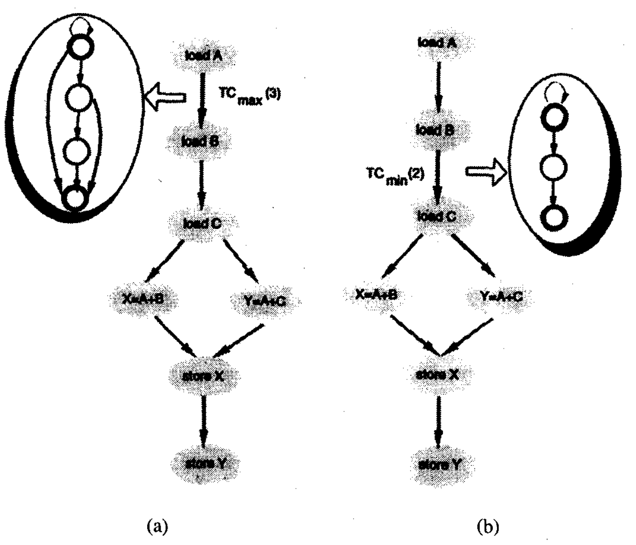


Fig. 12. Timing constraint example: (a) 3-cycle maximum timing constraint on **load A**; (b) 2-cycle minimum timing constraint on **load B**.

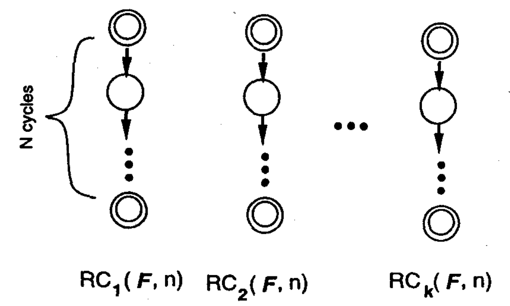


Fig. 13. General resource constraint automata.

all possible execution traces for **load B** to complete after the minimum of 2 cycles. □

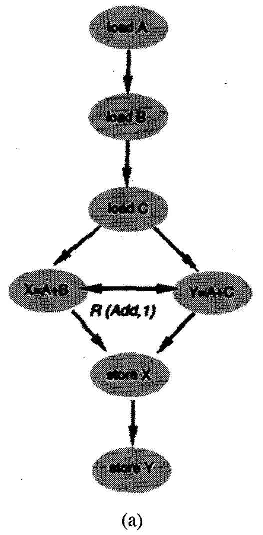
4.2. Resource Constraints

Constraining a design to a fixed number of resources can be modeled using automata. A resource is characterized by its type $\mathcal{F}$ (e.g., add, multiply, etc.), and the number of cycles n that it consumes. The maximum number of resources available at any time (the resource constraint) for type $\mathcal{F}$ is $R_{\mathcal{F},max}$.

The automaton that enforces a maximum resource usage of $R_{\mathcal{F},max}$ employs the product of k automata, where k is the number of operations of type $\mathcal{F}$. For each instance $j = 1, 2, \cdots, k$, a **comparator** automaton $RC_j(\mathcal{F}, n)$ is constructed. Since $\mathcal{F}$ is a n-cycle resource, RC_j is an automaton accepting n cycles, as shown in Fig. 13. $\mathcal{F}$ is being utilized by instance j when RC_j is not in the initial/final state and the output Out_j is assigned value 1.

Definition 10: The set of automata checking for resource constraints consists of k automata

$$RC_j(\mathcal{F}, n), \qquad j = 1, 2, \cdots, k$$



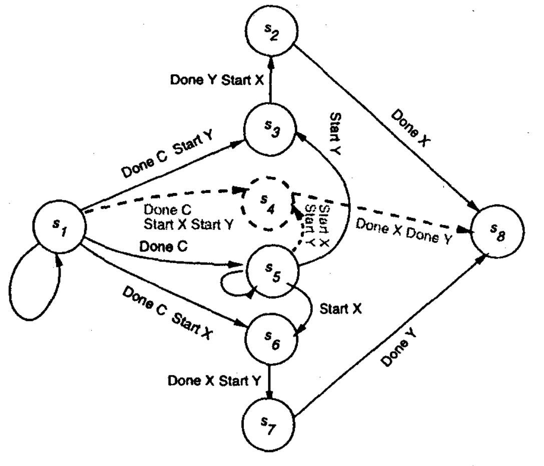


Fig. 14. Resource constraint example. (a) Insertion of $\mathcal{R}(add, 1)$ to enforce an one-adder resource constraint. (b) Product automaton for adder segment under constraint.

and a comparator function based on the output functions of RC_j:

$$R_{\mathcal{F},max} \geq Out_1 + Out_2 + \cdots + Out_k.$$

The output of the comparator is 1 if the inequality is satisfied, otherwise the output is 0.

The comparator output becomes part of the specification description. Indeed, functions $R_{\mathcal{F},max}$ is an instance of restriction function as introduced in Section 2.2. During implicit traversal of the product automaton, those traces that violate the resource constraint (therefore causing the output of $\mathcal{R}$ to be 0) are not constructed, leaving only those schedules that satisfy the constraint.

In the case where the resource $\mathcal{F}$ only requires one cycle, the constraint automaton degenerates to a combinational logic function.

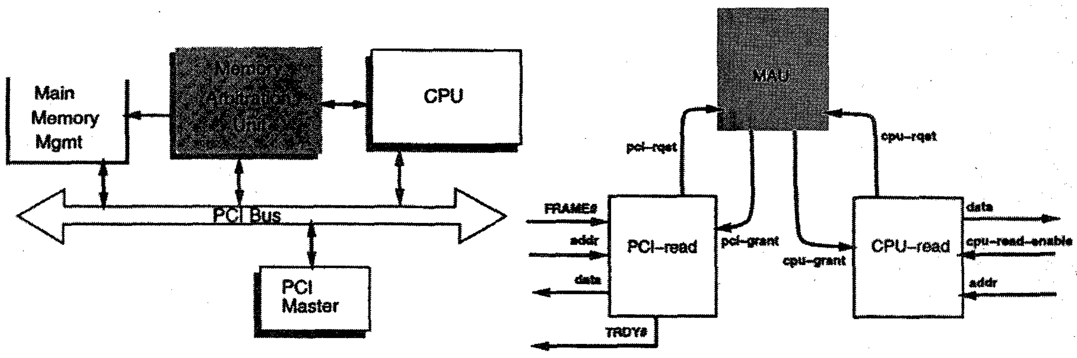


Fig. 15. Memory arbitration system and signals diagram.

```
mem_arbitrate (bus-rqst, bus-grant, cpu-rqst, cpu-grant ) {
    if (!cpu-rqst && !bus-rqst) {
        store cpu-grant = 0;
        store bus-grant = 0;
    }
    else if (!cpu-rqst && bus-rqst) {
        store cpu-grant = 0;
        store bus-grant = 1;
        while (bus-rqst) {
            if (cpu-rqst) {
                store bus-grant = 0;
                cpu_grant_memory (cpu-rqst, cpu-grant, bus-grant);
                store bus-grant = 1;
            }
        }
    }
    else {
        cpu_grant_memory (cpu-rqst, cpu-grant, bus-grant) ;
    }
}
```

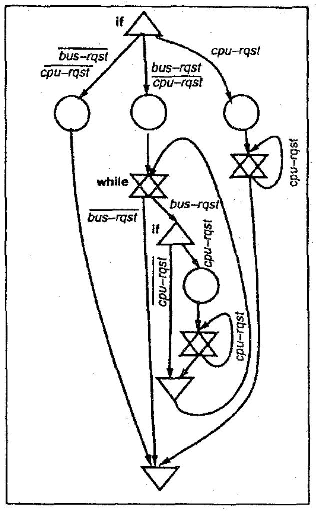


Fig. 16. Arbitration protocol.

Example 7: Consider the addition segment of Example 3. Suppose we place a resource constraint of one adder. A comparator expressed in Boolean logic is constructed allowing at most one adder to be utilized at any given time:

$$\mathcal{R}(add, 1) = \overline{Start_X} + \overline{Start_Y}.$$

Fig. 14(a) shows that $\mathcal{R}(add, 1)$ constrains the two additions. The resulting product automaton for the addition segment is shown in Fig. 14(b). Note that the previously *concurrent* path of $Start_X \cdot Start_Y$ is eliminated (shown in dotted lines). The comparator effectively enforces a mutual exclusion on the additions and *serializes* the operations. □

V. Synchronization Constraints

Sections III and IV have described basic constructs needed to model control flow and typical constraints. Using these, the automata framework can model most traditional high-level synthesis problems. In addition, our methodology supports modeling synchronization among interacting modules.

Most synchronization primitives can be modeled using the conditional and loop constructs previously described. As an example, the `wait` statement can be viewed as a degenerate case of a loop with no loop body and modeled accordingly.

An application domain that is especially suitable for our framework is the area of system-level design. Our method can be efficiently utilized to describe complex, control-oriented protocols. In this section, we highlight our methodology by focusing on a specific system-level example.

Example 8—Memory Arbitration: The design is part of a memory arbitration unit (MAU), whose function is to arbitrate accesses to main memory from the CPU (or its cache) and bus clients (or bus arbiters). In this case, we have chosen a PCI-bus[2] based system as the target architecture. The block diagram and the three major functional blocks with their input and output signals are shown in Fig. 15.

The goal is to design an arbitration unit that will grant memory accesses to either the CPU, or a PCI master device.

[2] *PCI Local Bus Specification Revision 2.0.*

```
pci_read (bus-rqst, bus-grant, frame, TRDY, addr, data) {
    store bus-rqst = 1 ; /* ask for the memory */
    wait (!FRAME) ;
    store TRDY = 1;
    store bus-rqst = 1;
    send_address (addr);
    while (!FRAME) {
        store TRDY = 0;
        receive_data (data);
        while (!bus-grant); {
            store TRDY = 1;
        }
    }
}
```

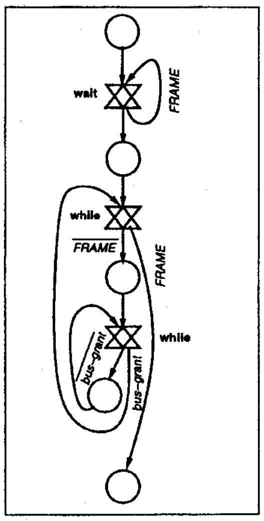


Fig. 17. *PCI-read* protocol.

```
cpu_read (cpu-read-enable, cpu-rqst, cpu-grant, addr, data) {
    wait (!cpu-read-enable);
    store cpu-rqst = 1 ; /* ask for the memory */
    while (!cpu-read-enable) {
        if (cpu-grant) {
            receive_data(addr, data); /* get data */
        }
    }
}
```

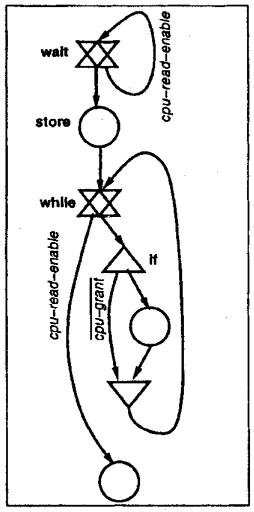


Fig. 18. *CPU-read* protocol.

For purposes of illustration, we use a simple scheme where a CPU request is always granted by preempting any existing bus accesses. Only *read*-cycles of the CPU and PCI are modeled. All accesses occur synchronously on the the system clock boundaries.

The arbitration protocol, PCI-read, and CPU-read protocols along with the automata (in CDFG notation) are shown in Figs. 16–18, respectively. The routine *cpu_grant_memory* sets `cpu-grant` to 1, `bus-grant` to 0, and waits for `(!cpu-rqst)`. The CPU-read is modeled as a generic read enabled by `cpu-read-enable`. The figures highlight control flow by showing conditionals and loops. Multiple operations are combined as single circles for illustration purposes.

The automata representations are built based on primitive constructs discussed previously. The memory arbitration *mem_arbitrate* unit is synchronized to *pci_read* and *cpu_read* through a set of synchronization signals: `pci-rqst`, `pci-grant`, `cpu-rqst`, `cpu-grant`.

Notice that in the specification of *mem_arbitrate*, there are no details of either the bus or CPU protocol. The arbitration protocol description is *generic*, while the protocol-specific portions of the design are described in *cpu_read* and *pci_read*. Changes in one module do not effect the description of the other two. Communication and synchronization takes place through the shared, synchronized signals. As an example, the bus protocol can be changed from PCI to another one without changing the arbiter or CPU descriptions, as long as the module has the same input/output signals. The automata formulations for each functional block can remain separate and modular until the product is formed. □

VI. Scheduling of the Specification Automaton

Our final goal is to derive a control-unit implementation from the specification automaton. In our context, such an implementation is a *state-table description* of the control FSM. An instance of the implementation is extracted from the specification automaton by examining the paths from initial state to final state. The set of all paths from initial to final state in the specification automaton constitutes all execution sequences that satisfy constraints imposed.

In general, there are many possible optimization cost criteria. Past approaches in using automata for sequential synthesis have focused mainly on state minimization e.g., [23]–[25], [14]. In this work, the optimization goal is to minimize the execution latency of the design. We assume that the clock period is a fixed parameter, and latency is defined as the number of clock cycles. The solution algorithms will be presented in the following two stages.

1) We focus on models (or submodels) whose latencies do not depend on input data, known as **fixed-latency** blocks. We demonstrate that the minimum latency schedule under constraints can be found using a shortest path algorithm. We then show how to derive a corresponding control unit.
2) We discuss **variable-latency** models, or designs whose latencies depend on input data. Generally, these include conditionals, data-dependent loops, and synchronization. We show that algorithms for the fixed-latency case can be used to generate control implementations with the least latency for any environmental condition.

6.1. Scheduling and Control Generation for Fixed-Latency Models

In this section, we present an algorithm that computes a schedule which minimizes the latency using the product automaton. From the schedule a state-table for the control FSM can be generated.

Despite the intractable nature of the scheduling problem under resource constraints, we show that the minimization algorithm is polynomial in the size of the product automaton. Therefore, the challenge lies in forming the automaton of reasonable size, since the problem size is not polynomially bounded. We rely heavily on the efficiency of BDD representation and implicit traversal procedure.

For fixed-latency designs, all paths in the product automaton correspond to feasible schedules that satisfy constraints. We recall Assumption 1, which says that each transition edge corresponds to one cycle. Since a shortest path necessarily contains the fewest transition edges and thereby the fewest clock cycles, we can state the following.

Proposition 1: Given an automaton representing the execution flows of a fixed-latency process, a shortest path from the initial to the final state determines a minimum latency schedule.

Example 9— Shortest Path: We reconsider now the instruction sequence of Example 7 under timing and resource constraints. Noting that the addition is a fixed-latency block, we show that the shortest paths constitutes a minimum-latency schedule.

Fig. 19 explicitly shows the relevant parts of the product automaton. There are two shortest paths, depending on the order of the addition operations: 1, 2, 5, 6, 7, 8, 9, 10, 13, 14 and 1, 2, 5, 6, 7, 8, 11, 12, 13, 14. Scheduling of operations

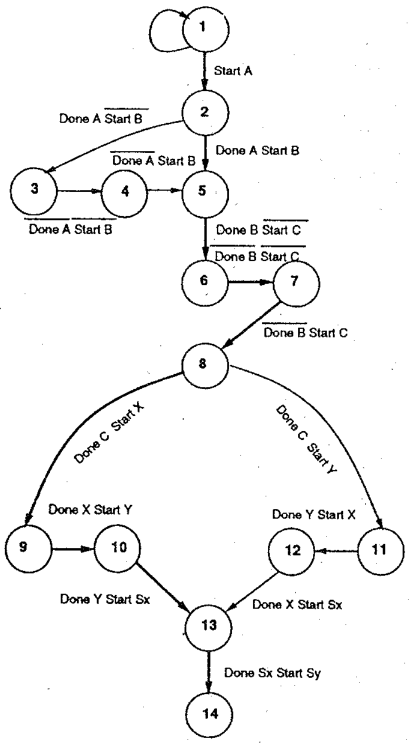

1
Start A
2
Done A $\overline{\text{Start B}}$
Done A Start B
$\overline{\text{Done A}}$ Start B
3
4
5
$\overline{\text{Done A}}$ $\overline{\text{Start B}}$
Done B $\overline{\text{Start C}}$
$\overline{\text{Done B}}$ $\overline{\text{Start C}}$
6
7
$\overline{\text{Done B}}$ Start C
8
Done C Start X
Done C Start Y
Done X Start Y
Done Y Start X
9
10
12
11
Done Y Start Sx
Done X Start Sx
13
Done Sx Start Sy
14

Fig. 19. Shortest path for addition example.

based on these shortest paths are minimum-latency since each operation is executed as soon as possible under constraints. □

Shortest Path Algorithm: The specification automaton can be seen as a graph where each edge has weight 1. Given a graph in explicit representation, any single-source shortest path algorithm (such as Dijkstra's algorithm [26]) can be used to solve the shortest path problem. Since the size of automaton is likely to be very large, an explicit representation and algorithm will be ineffective for large designs. In this section, we describe a shortest-path algorithm that can be applied in the *implicit traversal* framework.

The algorithm is divided into two steps: a *forward* and a *reverse* traversal. During forward traversal, the set of *next states* reachable in one transition are computed using an image computation in each iteration. States are assigned a label corresponding to the iteration in which they are *first* reached. The procedure iterates until no new states are found. During reverse traversal, the set of *previous states* that can be reached in one transition are computed using the inverse-image operator in each iteration. The shortest path is extracted by taking the state(s) with the highest label during the reverse traversal. The algorithm is described in Fig. 20. Note that the forward traversal can be seen as an extension of the implicit state traversal procedure [20], [21]. The only additional task required is labeling to state sets during traversal. The complexity of these two procedures are of the same order of the generic traversal.

The algorithm as shown generates one *explicit* minimum-latency schedule during the reverse traversal by choosing one path when more than one is available. The procedure can be modified slightly to *explicitly* generate *all* minimum-latency schedules. Instead of choosing one path, the procedure recursively follows all possible paths available. While the implicit automata modeling is able to efficiently represent and manipulate large number of paths, an *explicit* representation of all minimum-latency paths can require excessive storage since the implicit product automaton may contain exponential number of paths relative to number of states. Thus, enumerating explicitly all minimum-latency schedules should only be done when sufficient constraints have been applied to the product automaton using the implicit formulation. On the other hand, only one minimum-latency schedule is required for high-level synthesis applications, and it can be practically computed using the algorithm provided.

Example 10: Consider the automaton of Fig. 19. We perform the shortest path computation using $ForwardTraversal$ and $ReverseTraversal$. For the addition example, following the state labeling in Fig. 19, the run of the algorithm yields:

Forward Traversal	*Reverse Traversal*
$S_0 : \{1\}$	$S_9 : \{14\}$
$S_1 : \{2\}$	$S_8 : \{13\}$
$S_2 : \{3, 5\}$	$S_7 : \{12\}$ (picking 1 path)
$S_3 : \{4, 6\}$	$S_6 : \{11\}$
$S_4 : \{7\}$	$S_5 : \{8\}$
$S_5 : \{8\}$	$S_4 : \{7\}$
$S_6 : \{9, 11\}$	$S_3 : \{6\}$
$S_7 : \{10, 12\}$	$S_2 : \{5\}$
$S_8 : \{13\}$	$S_1 : \{2\}$
$S_9 : \{14\}$	$S_0 : \{1\}$

The resulting path can be traced by following the reverse traversal: it corresponds to the darkened path in the figure. □

6.2. Control Generation for Fixed-Latency Models

The implicit shortest path algorithm (Fig. 20) generates one minimum-latency schedule. From this schedule, a control implementation can be easily generated.

The output format of our control generation procedure is a *state-transition table*. The table can then be passed to conventional sequential optimizers e.g., [15]. To derive a state table representation, we need to consider different inputs and corresponding state transitions for each state.

Fig. 21 shows the pseudocode for generating the state table. Each entry in the state table is composed of a triple of present state, inputs (including conditionals), next state (s_1, w, s_2). Given a present state s_1 and a next state s_2, the input/conditionals required for the transition can be computed by:

$$w = (\chi \cdot s_1 \cdot s_2)|_{s_1 = s_2 = 1}$$

where χ is the characteristic function of the transition relation.

The procedure takes the minimum-latency schedule S as input, where S is the sequence of states in the specification automaton generated by algorithm $ReverseTraversal$. For each control step j, the state-transition table for S_j is constructed by taking all pairs of states in S_j and S_{j+1}, and computing w.

```
ForwardTraversal ($\mathcal{A}$) {
    $S_0$ = InitialState ($\mathcal{A}$);
    $T_0$ = FinalState ($\mathcal{A}$);
    TotalSet = $\emptyset$;
    $k = 1$;
    while ($T_0 \not\subset S_{k-1}$) {
        TotalSet = TotalSet $\bigcup S_{k-1}$;
        $S_k$ = image($S_{k-1}$) $- \bigcup_{j=0}^{k-1} S_j$;
        $k = k + 1$;
    }
    control_steps = $k - 1$;
}

ReverseTraversal($S$, control_steps) {
    $k$ = control_steps;
    while ($k > 0$) {
        $s_k$ = inverse_image($S_k$) $\bigcap S_{k-1}$;
        $k = k - 1$;
    }
}
```

Fig. 20. The implicit shortest path algorithm.

The complexity of this algorithm is $\sum_{j=0}^{n-1} S_j \times S_{j+1}$ number of computations of w, which is polynomial in the number of states.

Example 11: The state table for the addition example is:

state	**output / next state**
s_1	$Start_A$ / s_2
s_2	$Done_A \cdot Start_B$ / s_5
s_5	$Done_B \cdot \overline{Start_C}$ / s_6
s_6	$\overline{Done_B} \cdot \overline{Start_C}$ / s_7
s_8	$\overline{Done_B} \cdot Start_C$ / s_9
s_9	$Done_C \cdot Start_X$ / s_{10}
s_{10}	$Done_X \cdot Start_Y$ / s_{13}
s_{13}	$Done_Y \cdot Start_{S_x}$ / s_{14}
s_{14}	

The state table as is would terminate execution at s_{14}. In practice, if the process is repetitive, s_{14} is replaced by s_1. □

6.3. Control Generation for Variable-Latency Models

When considering models with conditional branching and iteration, latency may vary on the input data. Indeed, alternative paths of a conditional construct may require different latencies, and loops may have exit conditions that are data-dependent. Since synchronization `wait` statements are modeled as unbounded loops, they also give rise to variable-latency models.

When latency is data dependent, it is usually cumbersome to talk about minimum-latency schedules, because the latency itself depends on the input data. However, it is relevant to construct control units that use the least latency for any possible execution flow.

```
/* $S$: Sequence of states generated by $ReverseTraversal$
  $control\_steps$: Number of steps generated by $ForwardTraversal$ */
$GenStateTable$ ($S$, $control\_steps$) {
  for ($j = 0; j < control\_steps; j++$) {
    foreach $s_1 \in S(j)$ {
      foreach $s_2 \in S(j+1)$ {
        $P = s_1 \cdot s_2 \cdot \chi$;
        if ($P \neq NULL$) {
          $w = P|_{s_1=s_2=1}$;
          $EnterTable(s_1, w, s_2)$;
        }
      }
    }
  }
}
```

Fig. 21. Generating the state table from one schedule.

The algorithms outlined in Section 6.1 can be easily extended to find the least latency implementation for all input combinations, even in the presence of conditionals. The extension relies on the following three steps.

1) Sample and store the value determining the branch to be taken. Thus the state information of the specification automaton registers also the path along which each state was reached. The state of the specification automaton has then the form (c, s), where c records the condition input values, and s is the state of the original automaton.
2) Replace the final state f with the set F of all states of the form (c, f). Stop $ForwardTraversal$ when all such states are reached (i.e., when the original automaton has reached the final state for all values of the conditionals).
3) Apply $ReverseTraversal$ starting from F (i.e., find the shortest-path predecessor of each state for each branch of the conditionals).

The correctness of the procedure relies on a couple of observations. First, our modeling of *fork* and *join* retains only valid sets of nonmutually exclusive execution paths in the specification automaton as shown in Section 3.4. Such paths are the ones and the only ones that are explored by the implicit traversal procedure. Second, the implicit traversal procedure performs existential quantification on the conditional inputs, thereby insuring that *all* possible values of the input are included in the search. This leads to the following proposition:

Proposition 2: For variable-latency models, procedures $ForwardTraversal$ and $ReverseTraversal$ will generate the minimum-latency execution sequences for all conditional input combinations.

Example 12— Variable-Latency: Consider the following sample of code, with unit-delay operations:

```
V : L = M + N;
if (C)
 W : P = L + R ;
 X : R = P - Q ;
}
Z: store R
```

We first show the necessity of keeping the information related to the chosen branch. The automaton diagram (without branch information in the states) is shown in Fig. 22. Then, a shortest-path algorithm would reach the final state traversing only two edges, (i.e., along path (s_1, s_2, s_8)) and thus ignoring part of the computation (corresponding to the *true* value of the conditional.)

The complete automaton with the branch-taken information is shown in Fig. 23. Procedure $ForwardTraversal$ stops only when the final state set F (denoted in the figure as $\{F \cdot 8, T \cdot 8\}$) is reached by all possible alternative branches. Fig. 23 indicates also the control steps. One shortest-path path (corresponding to $\overline{C}$) has a latency of two, while another (corresponding to C) has a latency of four units.

The state table can be generated as in the case of fixed-latency models, while neglecting the branch state information, because the execution paths have already been identified. The state table for this example is the following:

input/state	output/next state
- / s_1	$Start_V$ / s_2
C / s_2	$Start_W$ / s_4
$\overline{C}$ / s_2	$Start_Z$ / s_8
- / s_4	$Start_X$ / s_6
- / s_6	$Start_Z$ / s_8
- / s_8	- / s_8

□

VII. Results

We have implemented a version of the automata framework and the scheduling algorithm described in this paper. We use *HardwareC* as our entry HDL to describe our processes and constraints. HERCULES is used to translate the description into a CDFG intermediate form known as SIF [16]. From SIF, the set of interacting automata is constructed and the product automaton is formed. The last step is to extract the shortest path which consists of the minimum-latency schedule. We show the effectiveness of our tool in two ways. First, we apply our algorithm to a set of conventional high-level synthesis

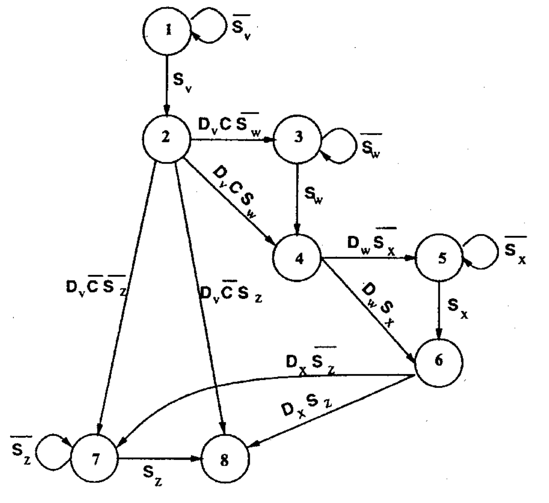


Fig. 22. Specification automaton.

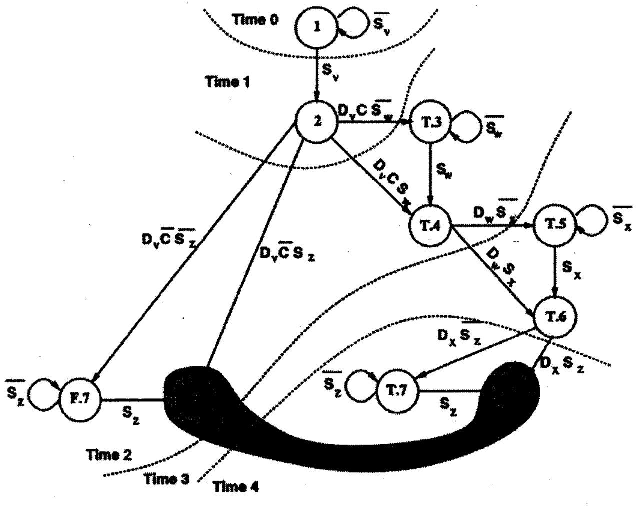


Fig. 23. Specification automaton with branch information. (Shaded states are final states.)

benchmarks to illustrate that our method is competitive for conventional applications. We then use the memory arbitration example and show the resulting controller implementation, which demonstrates an application that is easily modeled and synthesized using this framework.

We have run our algorithm on a set of standard high-level synthesis benchmarks. We impose resource constraints on selected benchmarks in order for our tool to solve more difficult instances of scheduling problems. The choice of a good variable ordering to form the product automaton is crucial to the feasibility of the method. It is possible to construct an effective ordering because the automaton structure is known *a priori*. In Table I, we show the time it takes to construct the product automaton and produce the minimum-schedule. The BDD size refers to the largest intermediate BDD encountered. The resource being constrained are included in parenthesis. The runtimes are in seconds on a DECStation 3000/400.

As an example, we vary resource constraints on *elliptic*. Table II shows the number of control steps as number of multipliers and ALU are varied. We assume that both ALU and multiplier take 1 cycle to complete. When we restrict the maximum timing constraint to less than 15 cycles (with no resource constraints), no feasible schedule exists.

TABLE I
RESOURCE CONSTRAINED SCHEDULING RESULTS

Benchmark	Control steps	BDD size	cpu(sec)
gcd	4	884	7.7
diffeq (1 alu)	5	3290	14.3
tseng (1 alu)	5	7299	41.6
parker86 (1 alu)	10	7704	67.1
elliptic (1 alu)	28	26725	512.0
ecc.encode	18	15438	133
ecc.decode	19	2788	79

TABLE II
ELLIPTIC SCHEDULE VARIANCES DUE TO CHANGES IN RESOURCES

multipliers	ALUs	Control steps	cpu(sec)
3	3	15	302.3
2	3	15	254.2
1	3	16	500.0
3	2	17	559.4
2	2	17	602.4
1	2	17	945.3
3	1	28	482.5

We compare the approximate runtimes of our `automata` method with those reported by Gebotys *et al.* [2] (`IP` formulation) and Radivojevic *et al.* [5] (`symbolic` formulation). The resulting number of control steps are the same for all methods since all three methods are exact. The average runtime for *elliptic* under various constraints is compared. The `IP` method (average runtime of 2 s) is fastest for this example, while the `symbolic` (average runtime of 400 s) and the `automata` (average runtime of 500 s) are comparable. Since `symbolic` and `automata` are symbolic, implicit techniques, they can potentially handle larger problem sizes. It is interesting to note that the `symbolic` method has a range of runtimes from less than 1 to 2000 s), while the range of runtime for the `automata` method is much smaller. This reflects the different treatment of constraint incorporation: the `automata` method is less sensitive to constraint variations.

These results are rough comparison only, as runtimes for all three methods were under different constraints and different machines. The main point is to note that all three methods can be applied to problems of practical size. `IP` and `symbolic` solutions require an upper bound to be given before starting, while the automata method does not.

The results of applying the `automata` method to the memory arbitration unit (MAU) example in Section V are reported in Table III. BFS is the number of steps required for the *ForwardTraversal* algorithm. The representation in BDD is very compact, as the state space required to model synchronization constraints is small. Our method deals with such control-dominated system-level designs very effectively. The low number of control steps that is typical in these designs

TABLE III
RESULTS ON THE MEMORY-ARBITRATION UNIT

Benchmark	BFS steps	BDD size	cpu(sec)
MAU	4	3164	14.3

works well in our framework, since there are fewer steps in the breadth-first traversal, thus keeping execution times low. Another advantage of the `automata` method is the ability of describing designs modularly. Different memory arbitration schemes can be substituted into *mem_arbitrate* without changing the other routines as long as the same I/O signals are used. Similarly, different bus and CPU implementations can be used as well.

VIII. Conclusion and Future Work

In this paper, we have demonstrated a novel way of using an automaton formulation for design representation and synthesis under general environmental constraints. In particular, we demonstrated that scheduling can be performed using environmental constraints (such as flexibility due to interaction with other components). These constraints are more general than the traditional resource/timing constraints.

We have shown that efficient state representation and traversal techniques can be extended to the high-level synthesis domain, in particular to scheduling. While we have reported experimental results on traditional high-level synthesis examples for the sake of comparison, our method is better suited for system-level designs. Design components can be described in a modular manner, with each component having its own automaton specification. The product is formed by synchronizing through a set of common communication signals, using the degrees of freedom among the interacting models.

There are a number of future research directions. First, the current discussion only addresses scheduling and control generation for *one* minimum-latency trace. One advantage of implicit enumeration scheme is that it is able to capture multiple traces and schedules efficiently. In such cases, it is possible to utilize the degrees of freedom of multiple solutions to optimize the control generation. In [12], a scheme to optimize nondeterministic FSM's using a similar framework is addressed. It attempts to find a best control implementation based on some cost factor (e.g., number of states). The optimization is performed on the entire set of possible execution traces, where implicit methods again is applied. Similar methods that leverage the preservation and further optimization of a *set* of feasible solutions are subjects of ongoing research. Second, a number of heuristics algorithms are possible to prune the search space and obtain faster runtimes. An idea under investigation is to employ the automata modeling and solution for interface protocols, and resort to traditional scheduling techniques for operations not involved in synchronization. Lastly, a number of techniques can be added to enhance the performance of BDD operations, including a generalized BDD variable ordering scheme as well as the use of alternate BDD representations (e.g., zero-suppressed BDD's).

Acknowledgment

The authors are indebted to J. Fron, who implemented the SIF to automata compiler, and to Dr. D. Ku for his comments. They also thank the anonymous reviewers for pointing out several imprecisions in the first draft of the paper.

References

[1] M. Garey and D. Johnson, *Computers and Intractability*. San Francisco, CA: Freeman, 1979.

[2] C. H. Gebotys and M. I. Elmasry, "Global optimization approach for architectural synthesis," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 1266–1279, Sept. 1993.

[3] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-level Synthesis: Introduction to Chip and System Design*. Norwell, MA: Kluwer Academic, 1992.

[4] D. Filo, D. C. Ku, C. N. Coelho, and G. De Micheli, "Interface optimization for concurrent systems under timing constraints," *IEEE Trans. VLSI Syst.*, vol. 1, pp. 268–281, Sept. 1993.

[5] I. Radivojević and F. Brewer, "Symbolic scheduling techniques," in *IEICE Trans. Inform. Syst.* vol. e78-d, no. 3, Japan, Mar. 1995.

[6] A. Takach, W. Wolf, and M. Lesser, "An automaton model for scheduling constraints in synchronous machines," *IEEE Trans. Comput.*, vol. 44, pp. 1–12, Jan. 1995.

[7] C. Coelho and G. De Micheli, "Dynamic scheduling and synchronization synthesis of concurrent digital systems under system-level constraints," in *ICCAD, Proc. Int. Conf. CAD*, 1994, pp. 175–181.

[8] R. Camposano, "Path-based scheduling for synthesis," *IEEE Trans. Computer-Aided Design*, vol 10, pp. 85–93, Jan. 1991.

[9] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. Cambridge, MA: MIT, 1988.

[10] C. A. R. Hoare, *A Model for Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[11] J. Fron, J. C.-Y. Yang, M. Damiani, and G. De Micheli, "A synthesis framework based on trace and automata theory," in *Int. Workshop. Logic Synthesis*, 1993 pp., 5c1–5c15.

[12] M. Damiani, "Non-deterministic finite state machines and sequential don't cares," in *EDAC Proc. Europ. Design Automat. Conf.*, 1994, pp. 192–198.

[13] G. Saucier, M. C. Depaulet, and P. Sicard, "Asyl: A rule-based system for controller synthesis," *IEEE Trans. Computer-Aided Design*, vol CAD-6, pp. 1088–1097, Nov. 1987.

[14] P. Ashar, S. Devadas, and A. R. Newton, *Sequential Logic Synthesis*. Norwell, MA: Kluwer Academic, 1992.

[15] E. Sentovich, *et al.*, "Sequential circuit design using synthesis and optimization," in *ICCD, Proc. Int. Conf. Comput. Design*, Oct. 1992, pp. 328–333.

[16] D. Ku and G. De Micheli, *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Norwell, MA: Kluwer Academic, June 1992.

[17] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw Hill, 1994.

[18] W. Thomas, "Automata on infinite objects," in *Handbook Theoretical. Comput. Sci.*, New York: Elsevier Science, 1990, pp. 133–191.

[19] Z. Kohavi, *Switching and Finite Automata Theory*. New York: McGraw-Hill, 1978.

[20] O. Coudert and J. Madre, "A unified framework for the formal verification of sequential circuits," in *ICCAD, Proc. Int. Conf. CAD*, Nov. 1990, pp. 126–129.

[21] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli, "Implicit state enumeration of finite state machines using BDD's," in *ICCAD, Proc. Int. Conf. CAD*, Nov. 1990, pp. 130–133.

[22] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors," in *DAC Proc. Design Automat. Conf.*, 1992, pp. 112–115.

[23] J. Kim and M. M. Newborn, "The simplification of sequential machines with input restrictions," *IEEE Trans. Comput.*, vol. C-21, pp. 1440–1443, Dec. 1972.

[24] J.-K. Rho, G. Hachtel, and F. Somenzi, "Don't care sequences and the optimization of interacting finite state machines," in *ICCAD Proc. Int. Conf. CAD*, 1991, pp. 418–421.

[25] S. Devadas, "Optimizing interacting finite state machines using sequential don't cares," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 1473–1484, Dec. 1991.

[26] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 1990.

Jerry Chih-Yuan Yang received the B.S.E.E. and M.S.E.E. degrees in 1990, and is currently working toward the Ph.D. degree in electrical engineering at Stanford University, Stanford, CA.

His research interests include logic synthesis; system-level modeling, verification and synthesis; and finite-state based synthesis methods.

Giovanni De Micheli (S'79–M'82–SM'89–F'89) received the Dr. Eng. degree in nuclear engineering in 1979 from the Politecnico di Milano, Italy, and received the M.S and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1980 and 1983, respectively.

He is an Associate Professor of Electrical Engineering and Computer Science at Stanford University. From 1984 to 1986 he was with the IBM T. J. Watson Research Center, Yorktown Heights, NY, where he was project leader of the Design Automation Workstation group. Previously, he held positions at the Department of Electronics, Politecnico di Milano, Italy, and at Harris Semiconductor, Melbourne, FL. His research interests include several aspects of the computer-aided design of integrated circuits with particular emphasis on automated synthesis, and optimization and verification of VLSI circuits. He is the author of *Synthesisis and Optimization of Digital Circuits*, (McGraw-Hill, 1994), coauthor of *High-Level Synthesis of ASICA Under Timing and Synchronization Constraints* (Kluwer, 1992), and coeditor of *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, (Martinus Nijhoff Publishers). He was also 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 1987.

Dr. De Micheli received the Presidential Young Investigator Award in 1988, the Best Paper Award in 1987 in IEEE TTRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and two Best Paper Awards at the Design Automation Conference, in 1983 and 1993. He is an Associate Editor of the IEEE TRANSACTIONS ON VLSI SYSTEMS and *Integration: The VLSI Journal*. He was technical and general chairman of the International Conference on Computer Design-ICCD in 1988 and 1989, respectively. He has served as member of the technical committtee of the ICCD, ICCAD, and DAC Conferences. He is the Program Chair of the DAC '96 Conference.

Maurizio Damiani received the degree in electrical engineering in 1987 from the University of Bologna, Italy. He received the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA.

He is an associate professor with the University of Padua since 1992. His research interests include formal modeling, synthesis and testing of sequential finite-state circuits and systems.

Dr. Damiani received an AEI Scholarship in 1988 and a Rotary International Fellowship Award in 1989.